A Program Slicer for LF

A THESIS

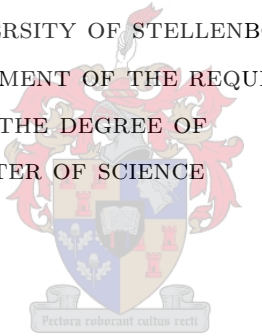
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

OF THE UNIVERSITY OF STELLENBOSCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE



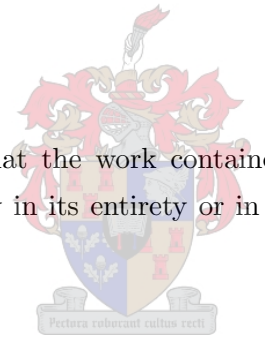
By

Françoise Louw

December, 2006

Supervised by: Mr J. J. Eloff

Declaration



I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: ..... Date: .....

Abstract

Program slicing was originally described by Mark Weiser in 1984. He proposed this as a technique to aid in debugging because he conjectured that this is what programmers did naturally when debugging. Here program slicing is applied to an experimental concurrent language called LF. Existing techniques are adapted to accommodate the unique features of the language.

Opsomming

Programdeling is oorspronklik deur Mark Weiser beskryf in 1984. Hy het dit voorgestel as 'n tegniek om ontfouting te vergemaklik, want hy het geglo dat dit is wat programmeerders van nature self doen. Programdeling word hier toegepas op 'n eksperimentele gelyklopende taal genaamd LF. Bestaande tegnieke word gewysig om die taal se unieke eienskappe in ag te neem.

Acknowledgements

I gladly acknowledge the help of several people who made this project feasible:

- Jacques Eloff for his support, guidance and endless encouragement.
- My family and friends for keeping me grounded.
- Leon Grobler, Erick Gerber and Hanno Bezuidenhout for all the answered questions.

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The static analysis of concurrent programs is more complex than that of sequential programs. This can be attributed to the fact that it is a field with a greater variety of problems due to the nature of concurrency. The programs are unpredictable, errors are usually not reproducible and execution traces started on the same input are hardly ever the same.

Slicing is a technique used to reduce a program by identifying and removing all statements that do not affect the value of a specified variable at a given point. It has been successfully applied to procedural languages for a number of applications. When originally conceived by Weiser [38], its main application was that of debugging. Slicing reduces the amount of code a programmer has to search through to locate the cause of an error. Weiser proposed that this is similar to the thought process of a programmer when debugging. Since then, slicing has been adapted to a wider variety of problems and applied in areas such as testing and verification [10, 37]. In the latter it is hoped that slicing will reduce the state space of a program when applied correctly, and increase the performance of the program verifier when checking a specific property.

1.1 The Subject of this Thesis

The goal of this thesis is the implementation of a static slicing tool for the LF language. This is accomplished by adapting existing techniques to deal with the unique characteristics of the language [33] that includes a synchronous communication model. The aim is to make the analysis as accurate as possible, although the precise analysis of

concurrent programs that consider synchronization as a constraint on execution order, is undecidable according to Ramalingam [30].

LF is an experimental concurrent language developed specifically with model checking in mind. It supports realistic implementations of typical embedded software such as device drivers, protocols and small user-level applications that can be executed on the actual target architecture. It is structured to only include features that will not complicate model checking unnecessarily. The LF project aims to model check executable code, reducing the additional risk of errors occurring when translating from a model to an implementation.

The slicing tool that is described in this thesis was developed using the Oberon language [39]. The choice to use Oberon as implementation language is motivated by the fact that the LF system[1] was implemented in Oberon. Secondary to this is the fact that Oberon encourages safe programming practises, the system incurs little overhead and the language itself had a significant influence on the development of the LF language.

1.2 The Outline of this Thesis

In Chapter 2, data structures for the abstract representations of programs that are useful with regard to slicing are discussed. An overview of current slicing techniques is presented and a few areas where slicing has been applied are also described. The influence of language characteristics is briefly investigated and the chapter concludes with a short discussion about LF.

The graph structure used by the LF Slicer as internal program representation is described in Chapter 3. A step-by-step discussion is presented for constructing each of the various structures found in the language. The discussions are accompanied by examples and the chapter is concluded with a larger example, to create a coherent picture.

In Chapter 4, the slicing algorithm is explained along with a discussion of the user interface. The implementation of the algorithm is described and illustrated with a few examples.

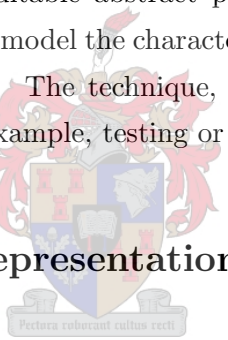
The results of the project, along with concluding remarks and suggestions for future work are presented in Chapter 5.

[1]The LF system primarily consists of the runtime environment, compiler, debugger and model checker.

Chapter 2

Background

There are a number of factors that govern the implementation of a program slicer, starting with the identification of a suitable abstract program representation. The chosen representation should be able to model the characteristics of the target language and fit the choice of slicing technique. The technique, in turn, will tie in with the intended application of the slicer, for example, testing or debugging.



2.1 Abstract Program Representations

Abstract program representations form an important part in the development of applications that manipulate programs at the source code level. To facilitate such manipulation, the application will transform the source code into an intermediate form that is better suited for analysis. Different abstract representations exist and each one lends itself towards a certain application such as optimization, transformation or slicing. In this section, structures that have been found useful in slicing are discussed.

2.1.1 Abstract Syntax Trees

An *abstract syntax tree* (AST) is a data structure constructed while parsing source code and is often used by a compiler as internal representation of a program's structure [2]. It is well-suited for optimization and source code regeneration. Every construct that a language might have, such as loop or selection statements, has a corresponding representation within the AST. This simplifies code generation and also defines a clear

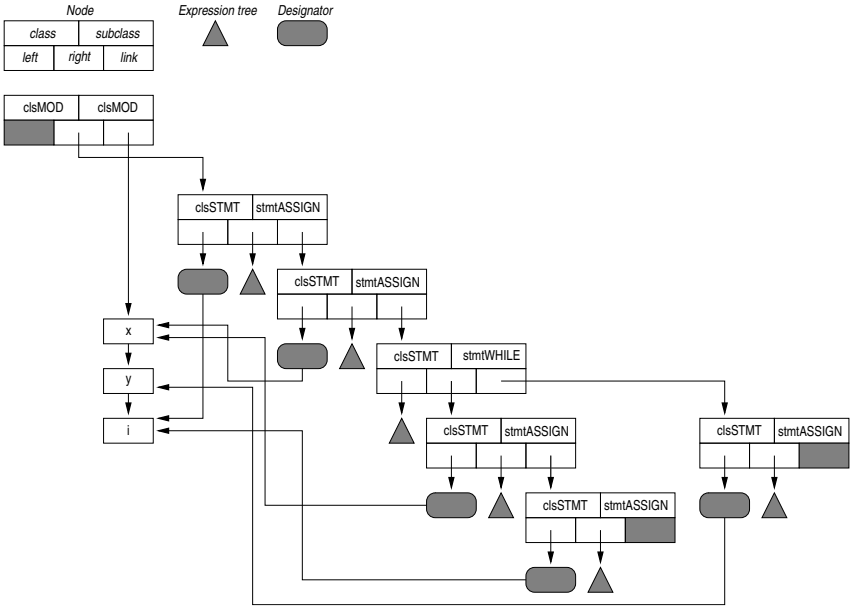


Figure 1: AST generated by the LF Compiler for the program in Listing 2.1.

interface between the front- and back-end of the compiler, allowing the re-use of the front-end when porting the compiler to a different hardware architecture.

ASTs do not have a fixed form, much the same way that different programming languages have different syntax for similar constructs. An `IF` statement is usually fairly recognizable within a language's particular syntax. Similarly, the representation of an `IF` statement in one version of an AST is usually comparable to and recognizable in the representation of another.

In Figure 1, an example of the AST generated by the LF compiler for the code in Listing 2.1 is given. Node 1 is the starting node for the AST and represents the encapsulating module. Its `right` edge leads to node 2 that represents the first statement in the body of the module (line 5).

The `left` edge of node 2 leads to a designator that is in turn connected to node 10 that represents the variable `i`. The `right` edge of node 2 is connected to an expression tree that is not given in detail in this example. The expression tree consists of designator nodes leading to variable nodes connected by operator nodes. The `link` edge of each assignment node leads to the following statement in the program, in this case, node 3 that represents line 6 of the source code. Node 4 represents the `WHILE` statement in line 7 with its `left` edge leading to the predicate expression. The `right` edge leads

Listing 2.1: A small LF program.

```
1  MODULE Example;
2  VAR
3    i, x, y : LONGINT;
4  BEGIN
5    i := 0;
6    x := 100;
7    WHILE i < 10 DO
8      x := x - i;
9      i := i + 1;
10   END;
11   y := x;
12 END Example.
```

to the statements within the body of the `WHILE` and the `link` edge to the statement following the `WHILE`.

2.1.2 Control Flow Graphs

A *control flow graph* (CFG) is a directed graph that represents the flow of control through a program. These graphs are mainly used for optimizations in compilers that make use of iterative data-flow analysis [24]. The nodes in a CFG can be either basic blocks[1] or individual statements. A directed edge between two nodes indicate a path in the program where the execution of the node at the tail of the edge will be followed by the execution of the node at the head of the edge. It is possible for a node to have multiple incoming and outgoing edges, representing multiple paths.

Consider the case where the nodes of a CFG represent individual statements. Then, formally, there exists a directed edge from statement S_1 to statement S_2 if S_2 immediately follows S_1 in some execution sequence. That is, if

1. there is a conditional or unconditional jump from statement S_1 to statement S_2, or
2. S_2 immediately follows S_1 in the order of the program and S_1 is not an unconditional jump.

We say that S_1 is a predecessor of S_2 and S_2 is a successor of S_1 [2]. Figure 2 gives an

[1] A basic block is a maximal sequence of instructions that can be entered only at the first instruction and exited only from the last [24].

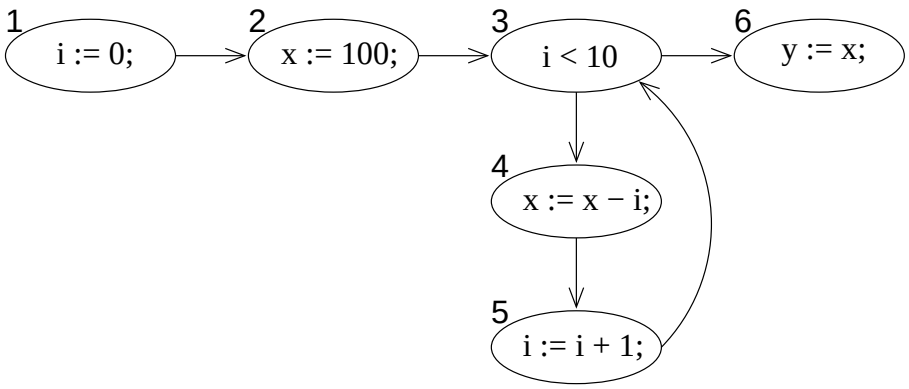


Figure 2: CFG of the program in Listing 2.1.

example of the CFG for Listing 2.1 where node 3 is a successor of nodes 2 and 5 and a predecessor of nodes 4 and 6.

2.1.3 Program Dependence Graphs

A *program dependence graph* (PDG) is an intermediate code form, often used for code optimization [4, 6]. The PDG consists of a *control dependence graph* (CDG) and a *data dependence graph* (DDG). Nodes in a PDG may be basic blocks, statements, individual operators or constructs at some intermediate level. For the purpose of slicing, PDG nodes typically represent statements because slicers usually operate at the statement level [24].

A CDG is a directed acyclic graph that has predicates as its root and internal nodes, and non-predicates as its leaves [24]. More formally, let $G = \langle N, E \rangle$ be a flow graph for a procedure, where N is the set of all nodes in the graph and E is the set of all edges. Node $m \in N$ post-dominates node $n \in N$, written m *pdom* n, if and only if every control flow path from n to the `exit` node passes through m. Node n is then control dependent on node m if and only if

1. there exists a control flow path from m to n such that every node in the path other than m is post-dominated by n, and
2. n does not post-dominate m.

A data dependence is a constraint that arises from the flow of data between statements. There are four types of dependencies:

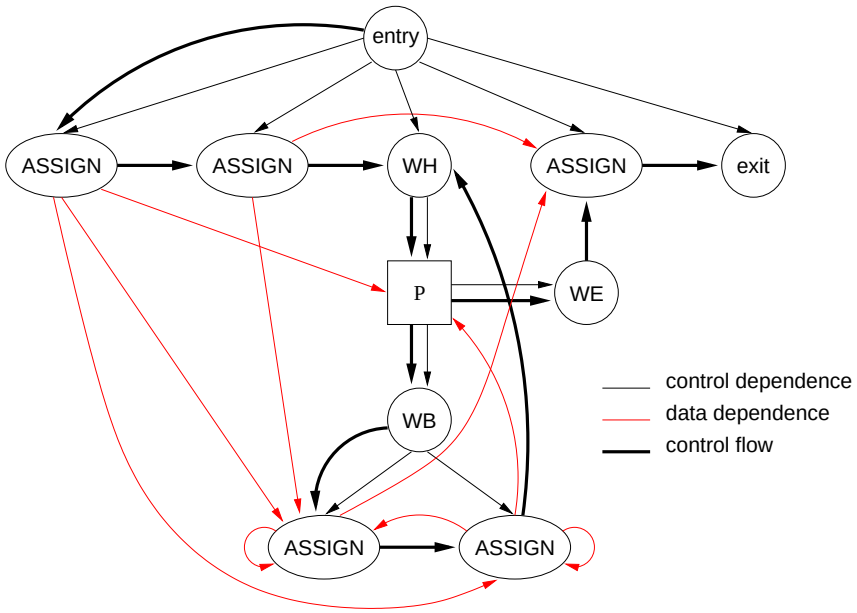


Figure 3: The PDG for the code fragment in Listing 2.1.

Flow Dependence If statement S_1 precedes statement S_2 and S_1 defines a variable that is later used by S_2, S_2 is flow-dependent on S_1.

Anti Dependence If statement S_1 precedes statement S_2 and S_1 uses some variable that is defined by S_2, there is an anti dependence between S_1 and S_2.

Output Dependence If statement S_1 precedes statement S_2 and both define the same variable, there exists an output dependence between S_1 and S_2.

Input Dependence If statement S_1 precedes statement S_2 and both use the same variable, there exists an input dependence between S_1 and S_2.

These data dependencies may be represented in a directed graph, labelling the edges with the type of dependency they represent. Traditionally flow dependence edges are not labelled [24] and in the case of slicing they are the only ones examined [14]. Figure 3 gives an example of a PDG.

2.2 Slicing Techniques

Weiser defines a slice as an executable program that is obtained from the original program by deleting zero or more statements [38]. A slicing criterion serves as input

to the slicer, indicating which variables at a given point in the program are of interest. It consists of a pair (n, V) where n is a node representing a statement in the abstract representation of the program P and V is a subset of P's variables. A slice S, composed of a subset of P's statements, is legitimate with respect to (n, V), if it halts on all inputs for which P halts and computes the same values for the variables in V each time that n is executed [34].

Slicing can be performed backwards or forwards. When slicing backwards, the slice extracted will contain all the statements in the program that could influence the value of the variable(s) at the given point in the program. If a forward slice is computed, the slice will contain all the statements in the program that may be influenced by the variable(s) defined by the statement specified in the slicing criterion. The focus of this thesis is on backward slicing and more refined forms of this method will be discussed briefly in the following sections.

2.2.1 Static Slicing

When calculating a static slice, only statically available information is used. All calculations are performed on some form of program representation such as an AST or a PDG. The program is never executed and consequently all possible inputs are considered as no assumptions can be made regarding the value of a variable at any time. All possible branch points that can influence variables in the slicing criterion will become part of the slice. This could lead to slices that are unnecessarily large and therefore not truly useful and can be addressed by taking a less conservative approach. This is achieved by computing additional information to reduce the extent of assumptions that would otherwise be made. Instead of immediately including a statement or branch point if it could possibly have an influence, more detailed analysis may reveal whether it is indeed possible. An area where this plays a great role is in the data-flow analysis of global variables where the trade-off between the extra calculations and the knowledge gained does not always make it a viable proposition.

The concept of static slicing was first introduced by Weiser [38] who used a CFG as program representation. Data flow equations were computed iteratively over the graph and used to determine both control and data dependencies between all statements.

For example, consider the DEF and REF sets in Table 1. They are the sets of variables that are defined and referenced respectively in the statements of the code fragment in

Listing 2.2: Product/Sum example.

```
1  read(n);
2  i := 1;
3  sum := 0;
4  product := 1;
5  while i <= n do
6  begin
7    sum := sum + i;
8    product := product * i;
9    i := i + 1
10 end;
11 write(sum);
12 write(product);
```

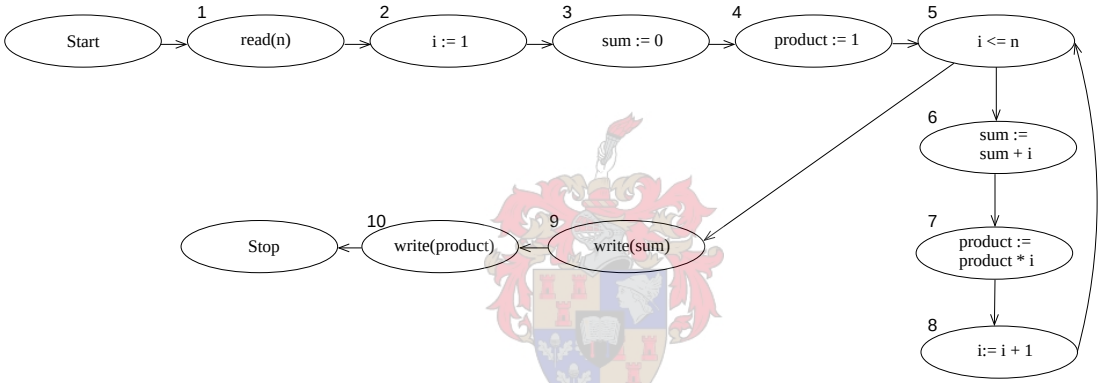


Figure 4: CFG of the Product/Sum example in Listing 2.2.

NODE	DEF	REF	INFL	R_C^0	R_C^1
1	{n}	∅	∅	∅	∅
2	{i}	∅	∅	∅	{n}
3	{sum}	∅	∅	{i}	{i, n}
4	{product}	∅	∅	{i}	{i, n}
5	∅	{i, n}	{6, 7, 8}	{product, i}	{product, i, n}
6	{sum}	{sum, i}	∅	{product, i}	{product, i, n}
7	{product}	{product, i}	∅	{product, i}	{product, i, n}
8	{i}	{i}	∅	{product, i}	{product, i, n}
9	∅	{sum}	∅	{product}	{product}
10	∅	{product}	∅	{product}	{product}

Table 1: Results of Weiser's algorithm for the example program in Listing 2.2 and slicing criterion (10, product).

Listing 2.2. The INFL set contains the statements that are control dependent on the node to which the set belongs. R_C^0 is known as the set of directly relevant variables (indicated by the superscript 0). This set is calculated by taking only data dependencies into account. From this set, a set of directly relevant statements, S_C^0, is calculated. These statements define a variable that is relevant at a control flow successor. The set of indirectly relevant variables R_C^{k+1} (the level of indirection is indicated by k), is determined by considering the variables in the predicates of the branch statements to be relevant if at least one statement in its body is relevant. The branch statements and R_C^{k+1} are used to form a set of indirectly relevant statements, S_C^{k+1}. The fixed point of this set gives the statements that form the slice. For further details see [34].

Ottenstein and Ottenstein [29] used a PDG that makes both the control and data dependencies explicit for each operation in the program [6], reducing slicing to a graph reachability problem that produces more accurate slices. This approach is also intuitively more appealing than Weiser's method, partly because it is easier to visualize and understand and partly because it does not require multiple iterations involving complex calculations and expensive operations.

2.2.2 Dynamic Slicing

Dynamic slicing differs from static slicing in that specific inputs are considered, producing a slice that is relevant to a single execution path. This is accomplished by executing the program to obtain an execution trace that is used when calculating the slice. The computation of the slice differs in that statements following the program point in the slicing criterion are also considered.

The approach of Weiser for calculating data flow equations iteratively over a control flow graph was extended by Korel and Laski [16] to accommodate dynamic slicing. This method may lead to unnecessarily large slices. A graph reachability approach was followed by Agrawal and Horgan [1]. They defined different approaches using variations of program dependence graphs to compute slices of increasing accuracy.

2.2.3 Conditioned Slicing

Conditioned slicing bridges the gap between static slicing where no input is specified and dynamic slicing where the input is fully specified. The slicing criterion is extended by adding a condition that describes the relation between different variables.

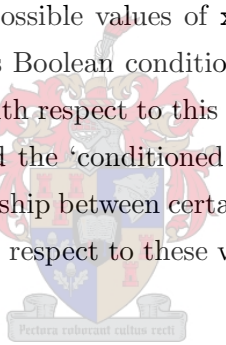
Listing 2.3: Conditioned slicing example.

```
1 scanf("%d", &x);
2 scanf("%d", &y);
3 if (x > y)
4   z = 1;
5 else
6   z = 2;
7 printf("%d", z);
```

Listing 2.4: Amorphous slicing example.

```
1 for(i=0, sum=a[0], biggest = sum; i < 19; sum = sum + a[++i])
2   if (a[i+1] > biggest)
3     biggest = a[i+1];
4 average = sum/20;
```

Consider the C program fragment in Listing 2.3. A Boolean expression, for example, `x == y+4`, can be used to relate the possible values of `x` and `y`. When the program is executed in a state that satisfies this Boolean condition, the assignment `z = 2` will never execute. Any slice constructed with respect to this condition may therefore omit this statement. This approach is called the 'conditioned approach' because the slicer has additional knowledge of the relationship between certain variables and therefore the state that the program must be in with respect to these variables when executing [11].



2.2.4 Amorphous Slicing

The techniques discussed in Sections 2.2.1, 2.2.2 and 2.2.3 have been both semantically and syntactically preserving. The aim of amorphous slicing is to preserve the semantics of a program, but not necessarily the syntax. Therefore additional static transformation techniques may be applied to aid in the simplification of a slice. This emphasizes the semantics of the program with regard to the slicing criterion.

The program fragment in Listing 2.4 correctly locates the largest element of the array `a` and stores the result in `biggest`. However, taking a traditional static slice that preserves syntax on the final value of the variable `biggest` does not make this very clear. This is because all that can be achieved through statement deletion is the removal of the final statement. By contrast, amorphous slicing on the final value for the variable `biggest` transforms the program to

```
for(i = 1, biggest = a[0]; i < 20; ++i)
  if (a[i] > biggest) biggest = a[i];
```

From the amorphous slice, where both statement deletion and program transformation techniques were used, it is far clearer what the goal of the original program fragment was. Notice how amorphous slicing retains the semantic guarantee that the program and slice behave identically with respect to the slicing criterion. Only the syntactic properties differ between traditional (syntax preserving) and amorphous slicing [11].

2.3 Applications of Slicing

Slicing has been applied to a variety of areas including debugging, testing and verification. A brief overview of these applications is presented in this section.

2.3.1 Debugging

When Weiser first introduced the concept of slicing in 1979, he held forth the idea that slicing was something programmers did naturally when debugging software [38]. When trying to locate the source of an error, a programmer tries to isolate the code that could affect the value of the variable(s) in question.

A slice taken at the point where the variable has an incorrect value will contain the line of code that incorrectly affects the value of the variable. Only errors that are present in the original source code can be found in this way. Errors that occur through omission, such as the failure to initialize a variable, will not be identified by slicing. It has however been argued by Harman and Danicic that amorphous slicing can be used to aid in the detection of such errors [9].

The benefit of slicing in debugging is that it reduces the size of the code through which the programmer must search. Some errors only manifest themselves on specific inputs. In these cases, dynamic slicing can be used since it incorporates the input when calculating the slice [11].

2.3.2 Testing

Errors are often introduced during the modification of software, even though the goal may be to correct existing errors. Regression testing is a process whereby software is re-tested following modification to ensure that faults have not been introduced or uncovered as a result of the changes made. This process ensures that the modified software still performs correctly for all test cases. Calculating a forward slice from the modification point isolates the code that could be affected, thereby limiting the amount of code that must be re-tested [10].

Harman and Danicic proposed further ways in which program slicing could simplify testing by introducing the concept of an introspective form of a program with the aid of implicit computations. This enables a program to explicitly compute its own robustness and enables the user to slice on the last line of such a program with respect to a 'flag' variable `robust` that will be introduced when transforming the program into the introspective form [8].

2.3.3 Verification

State explosion is one of the most challenging problems faced by model checkers. It is therefore important that models contain as little irrelevant information as possible. A model should only include information relevant to the specific property that is being checked to enable the model checker to perform a full verification. One of the main problems with this approach is identifying code that is relevant to the property. A slice on criterion deduced from the property[2] should isolate the relevant code from which the model must be derived [13].

Since the LF project aims to verify executable machine code directly, it can benefit from slicing in a more direct way, because the slice itself will be verified and the need to derive a model is eliminated [33, 35].

According to Vasudevan and Abraham [36], slicing allows for strong preservation of properties. Informally this means that a property that holds for the states of a program will also hold for the states in the slice and vice versa. When using slicing in conjunction with a model checker that allows the specification of properties in temporal logic, the propositions within the property form the variables in the slicing criterion.

[2] expressed in terms of a program's variables

Whereas other abstraction techniques sacrifice completeness for tractability and generality, slicing does not. Slicing will preserve correctness with respect to a specific property as opposed to a generic class of properties [36].

For the purpose of model checking, the slices produced must be closure slices [37]. A closure slice is a slice that contains all the statements that could affect the slicing criterion. This ensures that the resulting program is functionally equivalent to the original and that properties that hold for the slice also holds for the original program.

Iwaihara et al. [15] proposed the use of program slicing for the formal verification of hardware specifications in VHDL (VHSIC[3] Hardware Description Language). In the examples given they find that a significant reduction in the state space is achieved by using slicing.

2.4 Language Characteristics

Different languages offer different mechanisms to programmers that they can use to solve problems. Procedures, functions, parameters of different natures and concurrency are but a few of these. These mechanisms usually offer the same functionality, but are implemented in different ways by various languages. For example, synchronization of concurrent programs can be supported in a multitude of ways, including monitors and message passing. For each of these different techniques a new approach has to be taken when slicing, thereby complicating the implementation of a program slicer. The different ways in which parameters can be passed to a procedure or process also require accommodation within the slicing algorithm.

2.4.1 Procedural Languages

Weiser's algorithm for computing static slices accommodates the notion of interprocedural slicing, but the algorithm produces unnecessarily large slices. This can be attributed to the fact that the calling contexts of procedures are not taken into account and leads to the inclusion of computationally unrealizable paths [38].

Horwitz et al. use a graph theoretical approach to slicing [14]. They base their analysis on the PDG and so reduce the problem to a reachability exercise. They extended the

[3]Very High Speed Integrated Circuit

PDG to incorporate procedures by creating separate dependence graphs for each procedure and connecting them through edges to form a *system dependence graph* (SDG) that is traversed backwards. This means that once inside a procedure dependence graph, it is possible to follow edges back to different call sites, instead of the one from which the procedure was entered. This is referred to as the calling context of the procedure. The problem is solved by the introduction of transitive dependence edges between nodes in the procedure dependence graph representing parameters. To find these transitive edges, they make use of an attribute grammar, called a linkage grammar. They use this to model the call structure of each procedure and the intraprocedural, transitive flow dependencies among the parameter vertices of a procedure. Interprocedural, transitive flow dependencies among the parameter vertices of a system dependence graph are determined from the linkage grammar using a standard attribute-grammar construction: the computation of the subordinate characteristic graphs of the non-terminals of the linkage grammar. Their slicing algorithm performs a graph traversal in two phases. In the first phase no descent is made into procedures that are being called, i.e. , no traversal is made from call sites to the subgraph representing a procedure and in the second phase no ascent is made to call sites.

2.4.2 Concurrent Languages

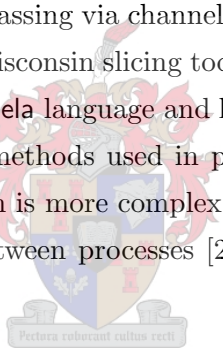
Concurrent languages differ from one another mainly in the way that they support the synchronization of concurrently executing processes and in the way they provide for communication between these processes. The most common ways for facilitating communication are shared variables and message passing over some type of channel. Semaphores, message passing and monitors can be used to establish synchronization.

When dealing with a concurrent language providing shared variables, an additional data dependence arises, namely interference dependence. A simple example of this is when a variable is defined in one thread and then accessed in another. When slicing such a language, one cannot simply traverse an interference dependence edge. This could result in the inclusion of unrealizable paths, thereby making the slice imprecise. Krinke [17, 19] has examined this problem and introduced the notion of a context, a threaded interprocedural witness and a valid execution path. He used these to find valid execution traces in a *threaded interprocedural program dependence graph* (tIPDG) and in the process diverted from a strictly graph reachability approach. The concurrency model he supports assumes that all threads are immediately active in the system, that

they do not share code, that all communication is done through global variables and that all statements are atomic and synchronized properly. It does not accommodate the dynamic creation of processes or communication via message passing. He does however make suggestions of how this solution can be adapted for Ada-style concurrency models [18].

Nanda and Ramesh [25] examined systems without explicit synchronization. They made use of co-begin and co-end routines and focused on loop-carried and loop-independent data dependencies with regard to global variables. They made use of a *threaded program dependence graph* (TPDG) similar to the one defined by Krinke. In addition to this, they use trace witnesses to identify valid execution paths in the graph and use this to determine valid slices.

Millett and Teitelbaum [21] focused on the slicing of Promela, a modelling language used by the Spin model checker. Promela shares many features with the LF language such as dynamic process creation, message passing via channels and polling communication. They extended the SDG used by the Wisconsin slicing tool to handle non-deterministic program constructs present in the Promela language and kept to a strictly graph reachability approach. They also adapted methods used in pointer analysis to solve their channel aliasing problem. This problem is more complex in Promela as it allows channels to be sent over other channels between processes [22]. LF does not provide this functionality.



Cheng [3] studied the slicing of concurrent programs and developed a graph called a *process dependence net* (PDN). This graph models the five dependencies in concurrent programs that are regarded as primary by Cheng. These are the usual control and data dependencies that one finds in sequential analysis, as well as selection, synchronization and communication dependencies. He uses these five dependencies to model the behaviour of a concurrent program and, as his representation is based on a graph-theoretical approach, it is language-independent. Once the PDN is constructed, the problem of slicing a program can easily be solved using a graph traversal algorithm, but the PDN does not support an interprocedural solution [34]. The examples given by Cheng are based on occam 2 code fragments.

2.5 The LF Language

The LF language [33] was developed to facilitate the formal verification of embedded software at the implementation level. This ensures that the code will perform correctly with respect to all properties that are checked, as the code that is verified is exactly the same as the code that will execute.

The LF syntax is based on that of the Oberon language [39]. LF is a concurrent language that supports processes that execute in parallel. It also supports run-to-completion processes that are similar to a procedure call. The decisions taken in the design of LF all have model checking as primary goal and to this end, all features that would negatively affect model checking were removed. This includes dynamic memory allocation.

LF processes communicate using built-in communication primitives. Channels are bi-directional and provide the communication medium between processes. They can be accessed globally or via parameters. Messages are passed to a channel using the send (`!`) primitive called a 'bang' and taken from a channel using the receive (`?`) primitive, otherwise known as a 'hook'. Communication is synchronous. LF also provides polling communication through the `SELECT` statement. This allows a process to poll different communication symbols on a specific channel and/or monitor several channels at a time. The different polling statements of a `SELECT` are called `WHEN` statements and a `SELECT` will suspend the process until one of its `WHEN` statements evaluates to `TRUE`.

Channels must be instantiated before they can be used. This is done by using the intrinsic process `NEW`. When a channel is `NEW`ed, various queues will be created within the runtime system that manage the messages passed on the channel [7]. The use of `NEW` is restricted to channel variables as dynamic memory allocation is not supported.

The `CREATE` keyword is used to instantiate a process that executes in parallel with its parent. The parent process will only terminate once all the children instantiated by it have terminated. Calling a process instantiates a process that will cause the parent to block until the child has terminated.

Listing 2.5 shows an example of a LF program. Two processes, `P0` and `P1` are created, one with a channel parameter and one without. The processes communicate with each other over a channel (`chan`) using send (`!`) and receive (`?`) primitives before terminating. Process `P0` initiates the communication by sending a message over the channel (line 13) that is received by `P1`. The content of the message is automatically

Listing 2.5: Example of LF syntax.

```
1  MODULE Example;
2
3  TYPE
4    chanDesc = [a(LONGINT), b(char)];
5
6  VAR
7    chan : chanDesc;
8
9  PROCESS P0;
10 VAR
11   x : CHAR;
12 BEGIN
13   chan ! a(5);
14   chan ? b(x)
15 END P0;
16
17 PROCESS P1(comms : chanDesc);
18 VAR
19   y : LONGINT;
20 BEGIN
21   comms ? a(y);
22   comms ! b('e')
23 END P1;
24
25 BEGIN
26   NEW(chan);
27   CREATE P0;
28   CREATE P1(chan)
29 END Example.
```


stored in y when the ? statement is executed.

2.6 Summary

There exists a close coupling between the structure used as intermediate program representation and the characteristics of the language being sliced. Chapter 3 introduces the *communicating concurrent dependence graph* (CCDG) – an extension of the traditional PDG. The CCDG was designed for the LF Slicer to support the various characteristics of LF such as its model for synchronous communication and other control structures such as SELECT statements.

Chapter 3

The Communicating Concurrent Dependence Graph

The CCDG is used as abstract representation of LF programs. It incorporates programming constructs unique to LF, including its concurrency model and method for providing synchronous interprocess communication. The LF Slicer uses the CCDG to calculate static backward slices based on some criterion and, together with the AST generated by the LF Compiler, reproduces the source code of the slice. The algorithm for building the CCDG was adapted from a syntax-directed algorithm for constructing PDGs.

3.1 The Abstract Syntax Tree

The LF Compiler generates an AST that provides a clear interface between the front-end and back-end of the compiler. This makes it possible to reuse the front-end for different tools, including a program slicer.

The tree serves as input to the graph building component of the slicer and provides enough information about the original source code to allow regeneration thereof. Links are established between nodes in the CCDG and nodes in the AST so that the source code of the calculated slice can be generated by traversing the AST with the CCDG indicating which statements to discard.

3.1.1 Data-flow Analysis

Data-flow analysis is performed by the compiler on the AST after its construction. Reaching definitions are calculated using the equations set out in [2] with methods based on that of [31]. The reaching definitions are added to the nodes of the AST and used when constructing the CCDG to add data dependence edges. No attempt is made to make global variables context-sensitive, therefore every definition of a global variable can reach every use of a global variable. Techniques to address this issue are described in Section 5.3.1.

3.2 PDG Construction

Harrold and Rothermel proposed a syntax-directed technique to construct a PDG [12]. The AST is traversed in-order and each corresponding PDG structure is built as its AST counterpart is encountered in the tree. Their technique provides for basic assignments, `if` statements, `while` loops and various structured transfers of control such as `break`, `continue` and `goto`. Statements in their AST always have a fixed structure. For example, an `if` statement always has an `if` and an `else` clause present, even if the statement body of either clause is empty.

A PDG consists of a *control dependence subgraph* (CDS) and a *data dependence subgraph* (DDS) as described in Section 2.1.3. The algorithm builds the CDS from the AST and calculates GEN/KILL sets for each node in the CDS. From these sets data dependencies are derived and edges representing them are added to the CDS to form the DDS.

`IF` and `WHILE` statements had to be modified for application to `LF`. This is discussed in Section 3.3.1. In Sections 3.2.1 and 3.2.2 their construction using the PDG algorithm is discussed.

3.2.1 If Statements

An `if` statement in the PDG starts with a node representing its predicate. This node has two outgoing control dependence edges connecting it to two region nodes representing the TRUE and FALSE branches. Access to the nodes inside the body of either branch is governed by the respective region nodes. A stack is used to manage nested regions of control.

3.2.2 While Loops

A `while` loop begins with a region node (node 2 in Figure 5) to which the last statement in the body of the loop returns. The next node in the structure will be a predicate node (node 3) that represents the guard of the loop. The predicate node will lead to a region node (node 4) that controls the body of the loop and an exit node (node 7) that leads to the statement following the loop (node 8).

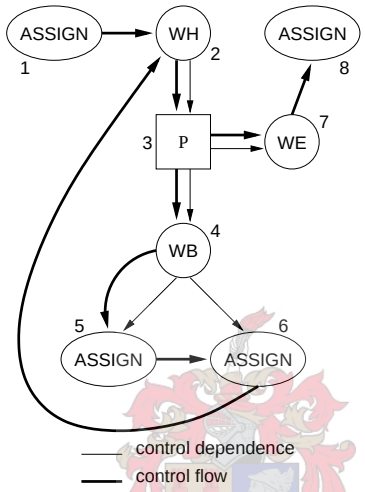


Figure 5: PDG of a `while` Statement.

3.3 CCDG Construction

The CCDG was developed to represent the unique features of LF. This aids the process of reducing static slicing to a graph reachability problem. To adapt the PDG construction algorithm for LF, some modifications and extensions had to be made. Statements not discussed in the original algorithm, but that exist in LF had to be incorporated and the differences between the AST of LF and the AST in [12] had to be accommodated.

A PDG represents a single thread of execution and does not support procedure calls. LF supports both concurrently executing and run-to-completion processes, the latter of which behave similarly to procedure calls. This led to the extension of the algorithm to construct a representation that closely resembles an SDG rather than a PDG. An SDG typically contains separate dependence graphs for each procedure and additional edges

are used to represent the dependencies between a call site and the instantiated procedure [14]. One could call such graphs representing individual procedures, procedure dependence graphs. In LF, processes will take the place of procedures.

LF supports synchronous communication through message passing via channels and additional nodes were introduced to model these channels and message flows. The flow of a message in the CCDG is indicated by a directed communication edge of which two types exist. Temporary communication edges join communication nodes with the global channel or channel parameter over which they communicate. These edges are later replaced by permanent communication edges that join a node with all its possible matching communication partners.

It is not possible to resolve the communication edges on the fly. This, as well as the resolution of channel connections to parameter nodes, require further processing after the initial construction. During this post-construction phase communication partners are found. These will be communication statements that share a channel, either directly (global channels) or indirectly (parameters), and use a matching symbol to communicate.

3.3.1 Modifications

Minor modifications were made to the algorithm to handle `IF` and `WHILE` statements. This is because the AST produced by the LF Compiler differs slightly in structure from the one used by the algorithm in [12].

`IF` Statement Only the branches of the `IF` that are present are included in the AST and it may not always contain an `ELSE` clause. Also, the `IF` statement in LF provides for an `ELSIF` clause. Another complication is the lack of a node indicating the end of a specific clause. One has to derive this knowledge from the fact that the next branch has started, and therefore the previous one must have ended.

`WHILE` Statement The `WHILE` loop in the AST of LF does not have a dedicated node indicating the end of the loop. This is the main difference between the loop structure in LF and the one provided for in the original algorithm.

LF does not provide any structured jumps such as `break` and `goto` statements, so the modified version of the algorithm excluded these.

3.3.2 Extensions

The algorithm was extended to include statements that are present in LF and not covered by the algorithm in [12]. Each extension will be discussed separately, giving the extract from the modified algorithm that is used to build it. Examples are also provided to clarify the discussion.

3.3.2.1 Process Declaration

Construction of modules and construction of process declarations is very similar. The difference occurs in the addition of parameters for the processes. The root node in any AST generated by the LF Compiler will represent the start of a module, and so lines 2 to 4 of Algorithm 1 will always start off the construction of a CCDG. An entry node is created by line 2 that represents the main region of control for the module body. During the recursive construction of the body of the module, both process declarations as well as code belonging to the module body will be encountered. Lines 7 to 16 of Algorithm 1 are responsible for the construction of a process.

When a process is encountered in the AST, an entry node for the process is created by line 7. This node is added to a list of entry nodes created for each process found during the traversal of the AST (line 9). This list aids in the resolution of edges from nodes representing process instantiations. As processes may have parameters, nodes representing them must be included. Lines 10 to 14 create a node for each parameter and adds it to the parameter list maintained for each entry node. After the nodes representing the start of a process and its formal parameters have been constructed, the body of the process is built recursively (line 15).

Figure 8 (on page 28) shows an example of a module containing one process declaration of which the entry node is node 2. This process has two parameters, `a` and `b`, represented by nodes 3 and 4. Node 1 is the entry node for the module.

3.3.2.2 The `CASE` Statement

The `CASE` statement is similar to a `switch` in C. The values for the expression in the header form the predicates of the different branches. The `CASE` statement also takes an optional `ELSE` clause to which it will default if none of the predicates are satisfied. In the CCDG, each branch of the `CASE` is preceded by a predicate which in turn is

Algorithm 1 Extract of CCDG Algorithm that builds processes.

```
1: 'module':
2: create an entry node for the beginning of the module
3: push the node onto the stack
4: initiate recursive call to build the body of the module
5:
6: 'process':
7: create an entry node for the beginning of the process
8: push the node onto the stack
9: add the entry node to the list of encountered processes
10: for all formal parameters do
11:     create a node representing the parameter
12:     add it to the entry node's parameter list
13:     add data dependence edges for the parameter
14: end for
15: initiate recursive call to build the body of the process
16: pop the entry node off the stack
```

followed by a region of control that governs all the statements within that branch.

When a `CASE` is constructed in the CCDG, the first node that is created is the `CASE` header node and it represents the expression that is evaluated. In Figure 6, this will be node 3 and it corresponds to line 6 in Listing 3.1. See lines 2–4 of Algorithm 2.

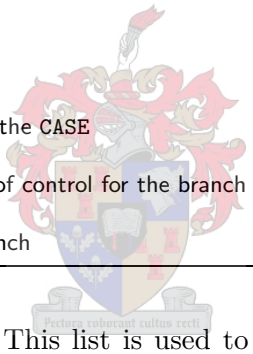
Listing 3.1: CASE statement.

```
1  MODULE Case;
2  VAR
3    x, y : LONGINT;
4  BEGIN
5    x := 5;
6    CASE x OF
7    1 : y := x + 1 |
8    2 : y := x + 2 |
9    ELSE
10     y := x;
11   END;
12 END Case.
```

The next step is to recursively build each branch of the `CASE`. This is initiated by line 6 whereupon lines 24–29 will execute. This section of the algorithm constructs the predicate governing each branch as well as the node representing the region of control. The body of the branch is also constructed recursively. Nodes 4 and 5 of Figure 6 represent the predicate and the region of control of the first branch of the `CASE`. Node 6 represents the statement contained within the branch and corresponds to line 7 of Listing 3.1. When the body of a branch is fully constructed and there is another branch after it, the last statement in the body must be added to the list of

Algorithm 2 Extract of CCDG Algorithm that builds the `CASE` statement.

```
 1: 'case':
 2: create a region node for the head of the CASE
 3: add data dependence edges
 4: push the node onto the stack
 5: for all branches of the CASE do
 6:     initiate recursive call to build next branch
 7:     if there is another branch or an ELSE then
 8:         add the most recently constructed node to the control flow unresolved list
 9:     end if
10:     pop the top of the stack
11: end for
12: if there is an ELSE then
13:     add a node representing the FALSE region of control for the ELSE
14:     push the node onto the stack
15:     initiate recursive call to build body of ELSE
16:     pop the top of the stack
17: end if
18: while the top of the stack is a predicate node do
19:     pop the top of the stack
20:     add to the control flow unresolved list if it does not have a both TRUE and FALSE labelled control
        dependence edge
21: end while
22: pop the case header off the stack
23:
24: 'caseblock':
25: create a predicate node for the branch of the CASE
26: push the node onto the stack
27: add a node representing the TRUE region of control for the branch
28: push the node onto the stack
29: initiate recursive call to build body of branch
```



nodes with unresolved control flow. This list is used to patch control flow jumps to nodes that have not yet been constructed as is the case with the last statement in a branch. This statement will have to jump to the first statement following the `CASE` for which the node has not yet been constructed at this point.

After all the branches and their bodies are constructed, all that remains is the construction of the `ELSE` if it is present. The details are covered by lines 12–17 of the Algorithm 2. Node 10 in Figure 6 represents the region of control governing the `ELSE`.

3.3.2.3 The `REPEAT` Statement

A `REPEAT` is a loop construct that terminates when its condition becomes `TRUE`. The evaluation of the condition occurs at the end of the loop, therefore the body of a `REPEAT` is always executed at least once. When building a `REPEAT` into the CCDG, a node that represents the region of control for the body of the loop is added. This is done by

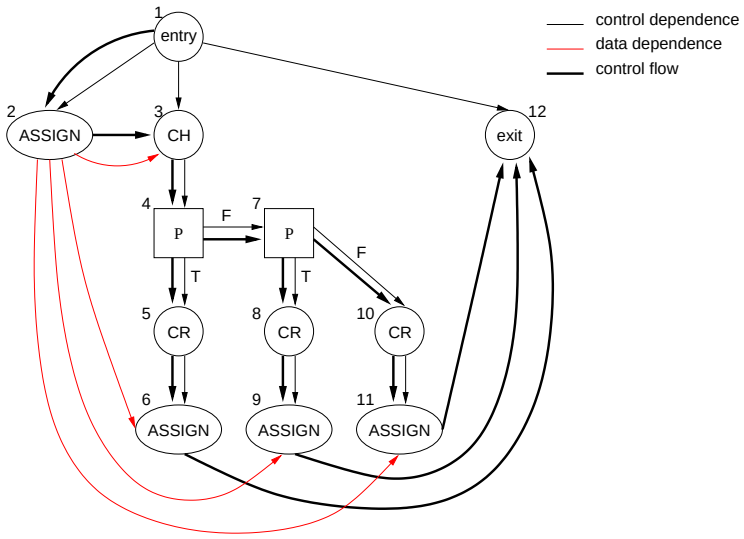


Figure 6: CCDG of the CASE statement in Listing 3.1.

lines 2–3 of Algorithm 3. The next step is to recursively build the body of the loop. When the body is completed, a node is added to represent the condition of the `REPEAT` (line 5). This node is then connected to the region controlling the body to form the loop structure (lines 6). Both control dependence and control flow edges must be added.

Algorithm 3 Extract of CCDG Algorithm that builds the `REPEAT` statement.

1: 'repeat':
2: create a region node for the body of the `REPEAT`
3: push the node onto the stack
4: initiate recursive call to build body of `REPEAT`
5: create predicate node
6: create the loop structure by adding control dependence and control flow edges between the body region node and the predicate node
7: add predicate node to the control flow unresolved list
8: add data dependence edges
9: pop the body region node off the stack

Figure 7 shows the CCDG that is constructed for the code in Listing 3.2. Node 4 is the region of control for the body of the loop and nodes 5 and 6 represent the statements inside the loop. Node 7 is the predicate of the loop and represents line 10 of the code.

3.3.2.4 The Call Statement

When a `CALL` statement is encountered in the AST, it requires the creation of a node that will represent it, such as node 9 in Figure 8. This is accomplished by line 2 of

Listing 3.2: REPEAT statement.

```
1  MODULE Repeat;
2  VAR
3    x, y : LONGINT;
4  BEGIN
5    x := 0;
6    y := 10;
7    REPEAT
8      y := y + 1;
9      x := x - 1;
10   UNTIL x >= 10;
11 END Repeat.
```

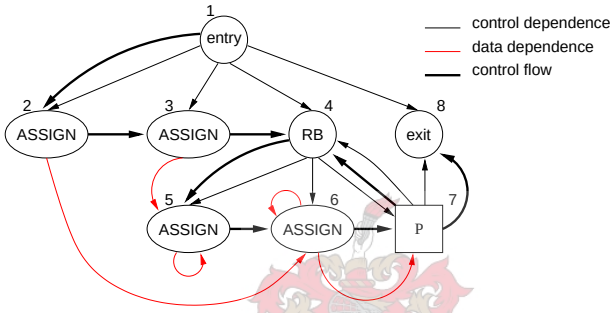


Figure 7: CCDG for the REPEAT statement in Listing 3.2.

Algorithm 4.

There is no reserved word `CALL` in the LF language. The use of a process name triggers a run-to-completion instantiation. The node representing the `CALL` statement is connected to the entry node of the instantiated process by way of a `CALL` edge. In Figure 8, node 2 represents the entry node of the process. An EXIT edge connects the exit node (node 6) of the process and the node representing the `CALL` statement. This is done to indicate the point in the parent to which the instantiated process will return. These edges are added in lines 13 and 14 of Algorithm 4.

A list of the entry nodes of all the processes that were found in the traversal of the AST is maintained so that their entry nodes may be found when references to them are made. LF does not support forward declarations so there is no need to address situations where a call or `CREATE` references an unknown process.

Parameters Processes may have parameters, and nodes representing each actual parameter must be created and attached to the node representing the call statement.

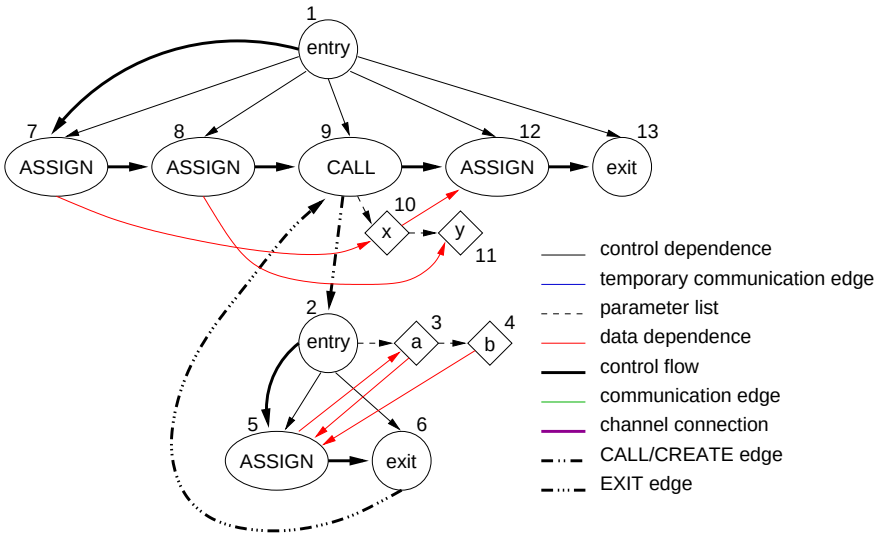


Figure 8: CCDG of the CALL in Listing 3.3.

Examples of these are nodes 10 and 11 in Figure 8. If one of the actual parameters is a channel, the corresponding formal parameter must be connected to the channel by a channel connection edge. If the channel node has not been created[1], an entry is made into the list of unresolved connection edges (lines 6–10 of Algorithm 4).

Algorithm 4 Extract of CCDG Algorithm that builds a CALL.

```
 1: 'call':
 2: create a call node
 3: for all parameters of the call do
 4:     create a parameter node
 5:     add the parameter node to the parameter list of the call node
 6:     if actual parameter is a channel then
 7:         add connection edges between the channel and the formal parameter of the process being instan-
            tiated
 8:     else if actual parameter is a channel parameter then
 9:         add parameter node to the list of unresolved channel connections
10:     end if
11: end for
12: add data dependence edges for the parameters of the call
13: add call edge between call node and entry node of process
14: add exit edge between call node and exit node of process
```

[1]The `NEW` statement that allocates memory for the channel has not yet been encountered

Listing 3.3: Example of a process call.

```
1  MODULE Call;
2  VAR
3    x, y, z : LONGINT;
4
5  PROCESS Add(VAR a : LONGINT; b : LONGINT);
6  BEGIN
7    a := a + b;
8  END Add;
9
10 BEGIN
11   x := 20;
12   y := 5;
13   Add(x, y);
14   z := x;
15 END Call.
```

3.3.2.5 The CREATE Statement

The CREATE statement represents the instantiation of a concurrent process. Nodes 14 and 16 in Figure 9 are examples of such nodes. This is accomplished by line 2 of Algorithm 5. The node representing the CREATE statement is connected to the entry node of the process being instantiated by a CREATE edge ((14, 2) and (16, 7)). An EXIT edge will be added between the exit node of the process and the node representing the CREATE. This is done to simplify the traversal of the graph and not to model the termination of the process as one cannot predict the execution point of the parent process when the child terminates. These edges are created in lines 13 and 14 of Algorithm 5.

Algorithm 5 Extract of CCDG Algorithm that builds a CREATE.

```
1:  'create':
2:  add a CREATE node
3:  for parameters of the CREATE do
4:      create a parameter node
5:      add the parameter node to the parameter list of the CREATE node
6:      if actual parameter is a channel then
7:          add connection edges between the channel and the formal parameter of the process being instantiated
8:      else if actual parameter is a channel parameter then
9:          add parameter node to the list of unresolved channel connections
10:     end if
11: end for
12: add data dependence edges for the parameters of the CREATE node
13: add call edge between CREATE node and entry node of process
14: add exit edge between CREATE node and exit node of process
```

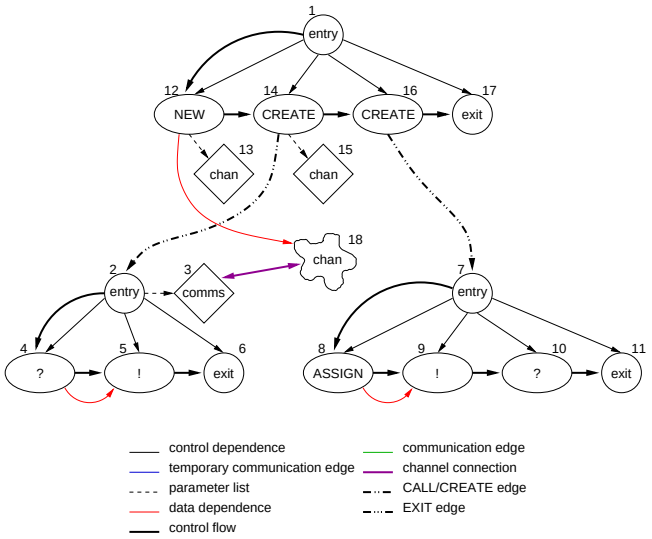


Figure 9: CCDG of the `CREATE`, `!` and `?` statements in Listing 3.4.

A list of entry nodes for all the processes that were found during the traversal of the AST is maintained so that they may be easily located when references to them are made. Forward references may be ignored as discussed in Section 3.3.2.4.

Parameters A node that represents each actual parameter must be created and attached to the node representing the `CREATE` statement. This is performed by lines 3–5 of Algorithm 5. If one of the parameters is a channel, the formal parameter corresponding to it must be connected to the channel with a channel connection edge. In Figure 9, the formal parameter is node 3 and it is connected to node 18 that represents the channel. If the channel node has not been created yet, an entry is made into the list of unresolved connection edges (lines 6–10 of Algorithm 5).

3.3.2.6 Communication Statements

A communication statement such as a `!` or `?` requires the addition of a node to the CCDG. An example is node 5 in Figure 9 which corresponds to line 12 of Listing 3.4. Temporary communication edges are created at this point that connect the statement to the channel or parameter via which the communication will take place. These edges are only temporary, because in a later phase after construction, they will be replaced by permanent communication edges to match `!` and `?` statements directly.

Listing 3.4: An example of a CREATE, ! and ? statement.

```
1  MODULE CREATEBANGHOOK;
2  TYPE
3    chanDesc = [a(LONGINT), b(LONGINT, LONGINT)];
4  VAR
5    chan : chanDesc;
6
7  PROCESS P0(comms : chanDesc);
8  VAR
9    x : LONGINT;
10 BEGIN
11   comms ? a(x);
12   comms ! b(x, 20);
13 END P0;
14
15 PROCESS P1;
16 VAR
17   y : LONGINT;
18 BEGIN
19   y := 23;
20   chan ! a(y;
21   chan ? b(y, z);
22 END P1;
23
24 BEGIN
25   NEW(chan);
26   CREATE P0(chan);
27   CREATE P1;
28 END CREATEBANGHOOK.
```

The difference between a ! and a ? is the direction of the communication edge. A ! will have an outgoing communication edge as it is sending a message, while a ? will have an incoming communication edge because it is receiving a message. The edge $(5, 3)$ in Figure 10 is an example of a temporary communication edge connecting a ! statement with a parameter. The edge $(18, 10)$ is an example of a temporary edge connecting a ? statement with a channel. These edges are added by lines 4–10 of Algorithm 6 in the case of a ! and lines 15–21 in the case of a ?. The conversion of temporary communication edges to permanent communication edges is discussed in Section 3.4.2.

3.3.2.7 The SELECT Statement

The conditional execution of ! and ? statements are allowed inside a SELECT. These statements are referred to as a polling ! and a polling ? because the SELECT allows one to poll communication on a channel with the aid of conditional expressions. This allows a process to monitor several channels at a time and react accordingly to the received

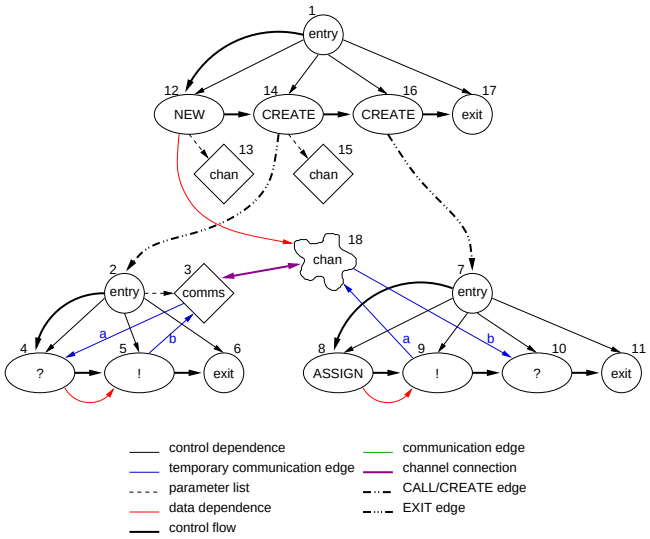


Figure 10: CCDG of the `CREATE`, `!` and `?` statements in Listing 3.4 with temporary communication edges.

input. It also allows a process to check for different symbols on the same channel.

A `SELECT` will poll each of its guards in turn, executing the first one that is enabled. The process executing the `SELECT` is suspended if none of the guards evaluated successfully and remains in this state until one of the guards is satisfied. The runtime system does not allow polling `!` and `?` statements in one `SELECT` to be matched with those inside another `SELECT`.

Algorithms 7 and 8 are combined to build the `SELECT` structure, where 7 is responsible for the main structure and 8 for the successive `WHEN` clauses.

The suspension of the `SELECT` is modelled with a loop in the CCDG to indicate that it will re-evaluate each `WHEN` clause if, on the first traversal, none of the guards were satisfied. This requires a header for the structure to be the anchor for the loop and is produced by lines 2 and 3 of Algorithm 7.

The next step is to build the polling communication statement for the first `WHEN` clause (lines 4 and 5). Additional predicates are handled by lines 6–10.

If the polling statement is a `!`, a release region is added. This controls the body of the `WHEN` clause and is called a release region because it is only at this point that the communication truly takes place. The node modelling the polling `!` only represents the possibility of the communication taking place. The release region is the point where

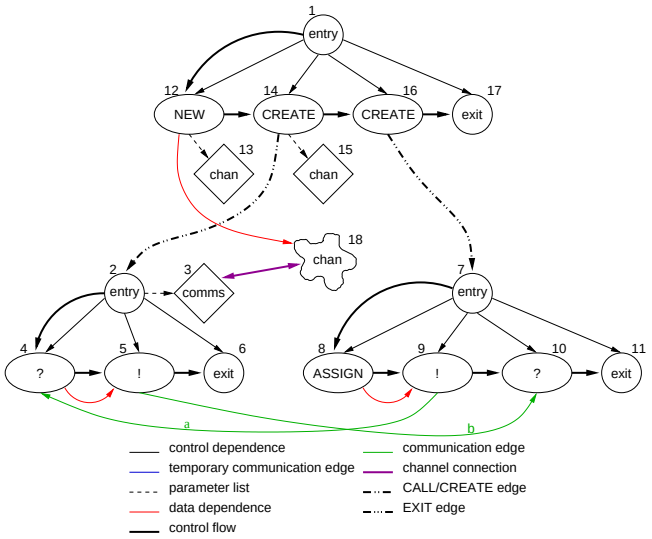


Figure 11: CCDG of the `CREATE`, `!` and `?` statements in Listing 3.4 resolved communication edges.

the communication does takes place. In the case of a polling `?`, the same reasoning applies, except that it will have a commit region. These nodes are added by line 11.

In Listing 3.5 process `P0` is created by the main body (line 30). This process contains a `SELECT` with three `WHEN` clauses. The main body executes a series of successive communication statements. Line 31 will match up with the second `WHEN` (line 17) as they have a corresponding channel (`c1`) and alphabet symbol. The value received into variable `x` on this line is multiplied by 3 before being sent over channel `c2`. It will be received by the `?` statement on line 32. That same value will then be sent on channel `c0` and will match with the first `WHEN` of process `P0` (line 14).

The `!` and polling `?` have a matching channel, matching alphabet symbol and the value of the variable satisfies the additional predicate of the `WHEN` clause. The value will be multiplied by 2 and returned on channel `c2`. Next, the main body will attempt to receive a value on channel `c1` (line 35). The only possible match for this statement is the polling `!` in the third `WHEN` (line 20) of process `P0`. After this communication takes place, the system will deadlock, because process `P0` is attempting to receive on channel `c2` (line 21) and there is no matching statement. Figure 12 shows the CCDG of the example in Listing 3.5 without any communication edges. Figure 13 displays the temporary communication edges and Figure 14 the permanent communication edges. The construction of these edges are described in Section 3.4.2.

Listing 3.5: SELECT statement.

```
1  MODULE SELECT;
2  TYPE
3    chanDesc = [a(LONGINT)];
4  VAR
5    c0, c1, c2 : chanDesc;
6    y : LONGINT;
7
8  PROCESS P0;
9  VAR
10   x : LONGINT;
11 BEGIN
12   WHILE TRUE DO
13     SELECT
14     WHEN c0 ? a(x) & x = 9 THEN
15       x := x * 2;
16       c2 ! a(x);
17     WHEN c1 ? a(x) THEN
18       x := x * 3;
19       c2 ! a(x);
20     WHEN c1 ! a(23) THEN
21       c2 ? a(x);
22     END;
23   END;
24 END P0;
25
26 BEGIN
27   NEW(c0);
28   NEW(c1);
29   NEW(c2);
30   CREATE P0;
31   c1 ! a(3);
32   c2 ? a(y);
33   c0 ! a(y);
34   c2 ? a(y);
35   c1 ? a(y);
36 END SELECT.
```

Algorithm 6 Extract of CCDG Algorithm that builds communication statements.

```
1: 'bang':
2: create a bang statement node
3: add data dependence edges
4: if channel is in parameter list then
5:     add communication edges between bang and channel parameter node
6: else if channel is a global channel then
7:     add communication edges between bang and channel node
8: else
9:     add the bang statement to the list of unresolved channel communications
10: end if
11:
12: 'hook':
13: create a hook statement node
14: add data dependence edges
15: if channel is in parameter list then
16:     add communication edges between hook and channel parameter node
17: else if channel is a global channel then
18:     add communication edges between bang and channel node
19: else
20:     add the bang statement to the list of unresolved channel communications
21: end if
```

3.3.3 Control Flow Resolution

Whenever control has to flow from a node to node that has not been created yet, an entry is made into the list of unresolved control flow edges. An example of where this would happen, would be `IF` statements. The last statement in a branch of an `IF` has to jump to the statement following the `IF`, but when that last statement is constructed, the node representing the statement following the `IF`, does not exist yet. There are three points during the construction where control flow is resolved:

1. Each time a new node is created, the control flow unresolved list is inspected. If it is not empty and the current region node on the stack is not a predicate node or a polling communication node, the outstanding edges between the nodes in the control flow unresolved list and the new node are added. The reason for not resolving edges when there is a predicate or polling communication node on top of the stack is because control flow cannot jump directly into another control structure such as an `IF` or `SELECT`. If the node in the control flow unresolved list is a predicate node, both control flow and control dependence edges must be added. For other nodes, only control flow edges are necessary. An exception to this rule is a predicate node representing a `REPEAT` condition. It only needs control dependence edges.

Algorithm 7 Extract of CCDG Algorithm that builds `SELECT` statements.

```
'select':
create a SELECT header region node
push the node onto the stack
create a polling communication node
push the node onto the stack
if there is an additional predicate then
    create a predicate node
    push the node onto the stack
    add data dependence edges for the predicate node
end if
add a commit or release region, depending on the type of polling communication
push the node onto the stack
add data dependence edges for the commit/release region
if channel is in parameter list then
    add communication edges between bang and channel parameter node
else if channel is a global channel then
    add communication edges between bang and channel node
else
    add the bang statement to the list of unresolved channel communications
end if
initiate recursive call to build body of first WHEN
if there is another WHEN then
    add most recently create node the the control flow unresolved list
end if
pop the commit/release region off the stack
for subsequent WHEN clauses do
    initiate recursive call to build WHEN
    if there is another WHEN then
        add most recently create node the the control flow unresolved list
    end if
    pop the commit/release region off the stack
end for
if there is a predicate node on top of the stack then
    add FALSE edges between the SELECT header and the predicate node as well as FALSE edges between
    the SELECT header and the polling communication node
else if there is a polling communication node on top of the stack then
    add FALSE edges between the SELECT header and the polling communication node
end if
pop all predicate and polling communication nodes off the stack
pop the SELECT header node off the stack
```

Algorithm 8 Extract of CCDG Algorithm that builds `WHEN` statements.

```
1: 'when':
2: create a polling communication node
3: if previous WHEN had an additional predicate then
4:     add FALSE edges between it and this communication node
5: end if
6: push the polling communication node onto the stack
7: if there is an additional predicate then
8:     create a predicate node
9:     push the node onto the stack
10:    add data dependence edges for the predicate node
11: end if
12: add a commit or release region, depending on the type of polling communication
13: push the node onto the stack
14: add data dependence edges for the commit/release region
15: if channel is in parameter list then
16:    add communication edges between bang and channel parameter node
17: else if channel is a global channel then
18:    add communication edges between bang and channel node
19: else
20:    add the bang statement to the list of unresolved channel communications
21: end if
22: initiate recursive call to build body of the WHEN
```

2. Another place where control flow edges are resolved is after the construction of the body of a `WHILE`. This is not discussed explicitly in the previous section as it is so similar to the adapted algorithm. However, all the nodes that are in the control flow unresolved list when the body of a `WHILE` has finished construction must be joined to the `WHILE` header node to form the loop structure.

3. Any nodes remaining in the control flow unresolved list when the exit node of a process or module is constructed will be joined with a control flow edge to the exit node.

3.4 Post Construction

The AST is traversed once and used as a guide to construct the CCDG. After the initial construction, channel connections must be resolved. This, as well as the permanent matching of communication partners, is performed in the post construction phase.

3.4.1 Channel Connections

When a process has a formal parameter that is a channel and that process is instantiated with a specific channel, a connection is established between the nodes representing the

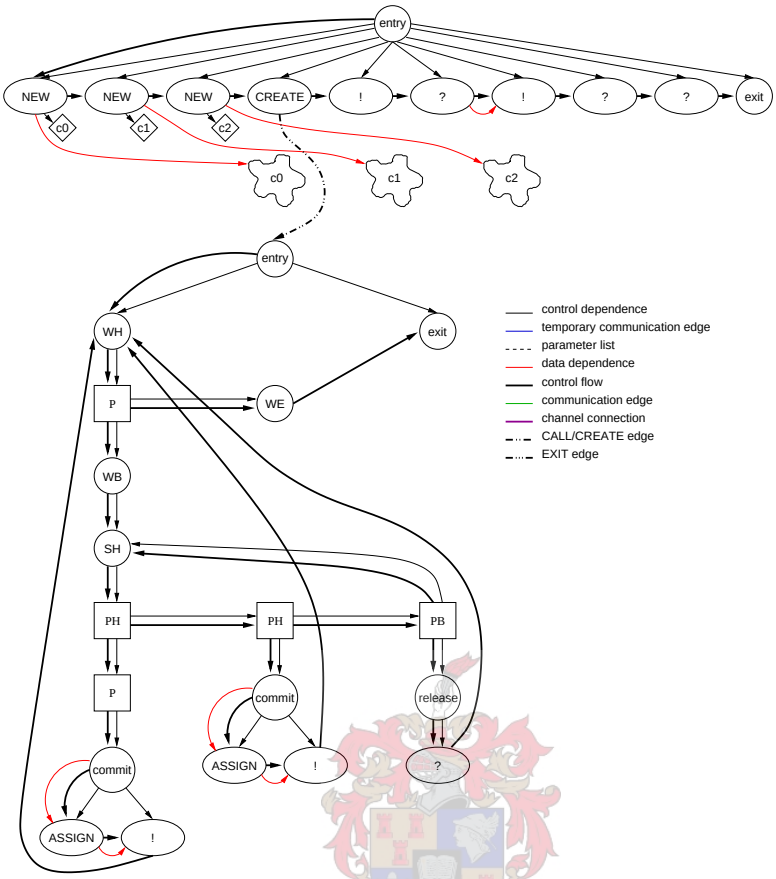


Figure 12: CCDG of the SELECT statement in Listing 3.5.

formal parameter of the instantiated process and the channel. For example:

```
MODULE MainProgram;
TYPE chanType = [a(LONGINT)];
VAR chan : chanType;
PROCESS P0(c : chanType);
BEGIN
  ...
END P0;

BEGIN
  NEW(chan);
  CREATE P0(chan);
END MainProgram.
```

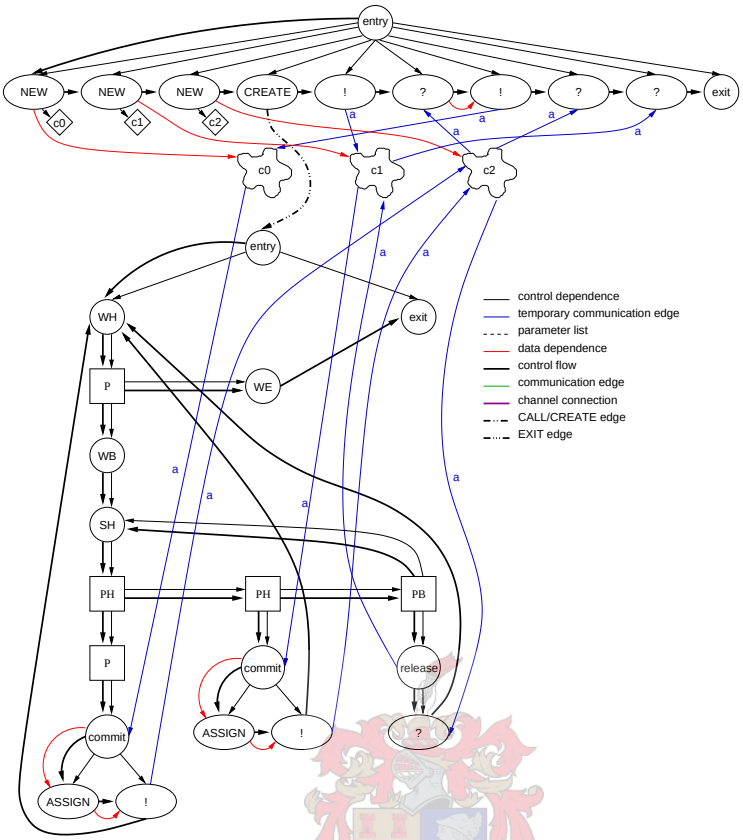


Figure 13: CCDG of the SELECT statement in Listing 3.5 with temporary communication edges.

It can happen that when the instantiation of a process takes place, the node representing the channel has not yet been created. In this case, an entry is made into the list of unresolved channel connections. Each time a `NEW` statement for a channel is encountered, this list is examined for outstanding connections so that they may be resolved.

Connections between channels and parameters are only established when an instantiation of a process is encountered in the AST. Therefore when the body of a process is being constructed in the CCDG, none of the instantiations of that process have been met yet, because LF does not allow for forward declarations.

If a process `P0` then instantiates another process `P1` and passes on one of it's formal channel parameters, the connections between the formal parameter of the instantiating process (`P0`) and the channels it aliases will not be finalized. For example:

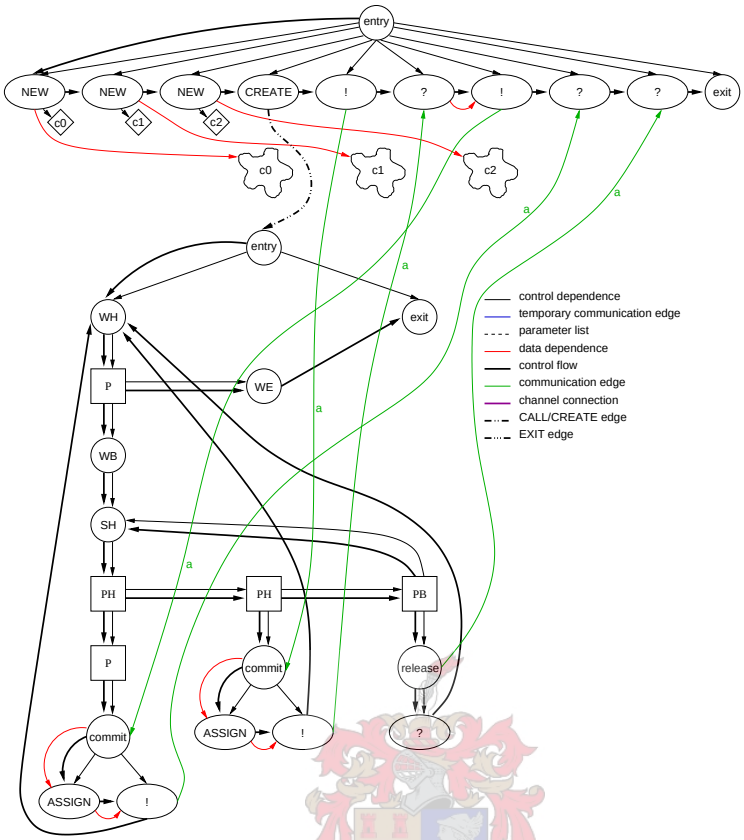


Figure 14: CCDG of the SELECT statement in Listing 3.5 with resolved communication edges.

```
MODULE MainProgram;
TYPE chanType = [a(LONGINT)];
VAR chan : chanType;
PROCESS P1(c : chanType);
BEGIN
  ...
END P1;
PROCESS P0(d : chanType);
BEGIN
  CREATE P1(d);
  ...
END P0;
BEGIN
  NEW(chan);
  CREATE P0(chan);
END MainProgram.
```

When `P0` creates `P1`, the main process has not yet created `P0`, so there is no connection yet between global channel `chan` and parameter `d`. It can therefore not be transferred to formal parameter `c` when `P1` is instantiated. An entry of the pair of formal parameters is made in the list of unresolved channel connections. During the post construction phase the connections of the one is transferred to the other.

3.4.2 Matching Communication Partners

Matching communication partners are `!` and `?` statements that share a channel, either directly via a global channel or indirectly via a parameter, and communicate using the same alphabet symbol. Polling communication statements cannot be matched with one another and communication statements within the body of a `WHEN` can only be matched with statements that succeed a communication statement that enabled the `WHEN`.

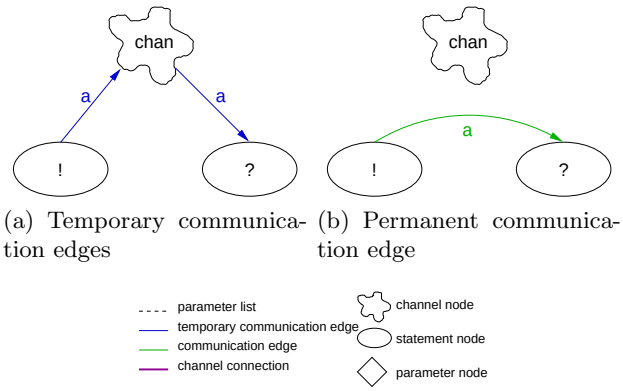
3.4.2.1 Non-polling Communication

The process of matching communication partners starts with non-polling communication statements. The possible message routes are traced from `!` statements to `?` and polling `?` statements. The `!` statements are automatically matched with `?` statements that share a common alphabet symbol if the `?` does not fall within the body of a `WHEN` clause. If it does, the path is noted as a possibility and confirmed at a later stage when polling communication is resolved. The `!` statements are also immediately matched with all polling `?` statements with whom they share a common alphabet symbol.

There are a number of ways in which a path between a `!` and a `?` can be established as shown in Figures 15–18. In Figure 15, both the `!` and the `?` communicate directly with the channel. This path is found by exploring the `!`'s outgoing communication edge as well as the outgoing communication edge of the channel.

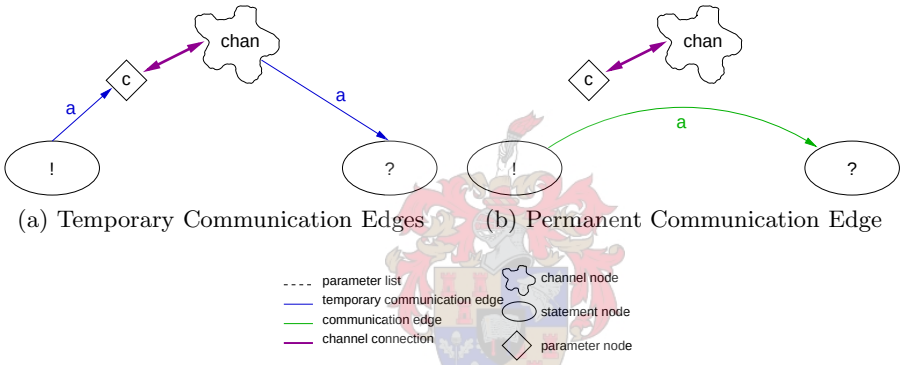
Figure 16 shows a `!` communicating via a parameter that is connected to a channel that is accessed directly by the `?`. To find this path, the outgoing communication edge of the `!` is explored, followed by the outgoing connection edge of the parameter, followed by the outgoing communication edge of the channel.

This situation can be reversed so that the `?` communicates via the parameter and the `!` accesses the channel directly as shown in Figure 17. The `!`'s outgoing communication edge, the channel's outgoing connection edge and the parameter's outgoing



(a) Temporary communication edges (b) Permanent communication edge

Figure 15: A ! and a ? communicating directly via a channel.



(a) Temporary Communication Edges (b) Permanent Communication Edge

Figure 16: A ! communicating via a parameter with a ? communicating directly via a channel.

communication edge will have to be followed to trace this path.

In Figure 18, both the `!` and the `?` communicate via parameters. To trace the path between them one must follow the outgoing communication edge of the `!`, the outgoing connection edge of the parameter, the outgoing connection edge of the channel and the outgoing communication edge of the parameter.

The aforementioned cases also hold for polling `?`s except when a `?` inside a `WHEN` is reached. The path would be found in the same way, but only confirmed at a later stage.

In summary, the following edges must be explored to find a path between a `!` and a `?`:

1. The outgoing communication edge for each `!` statement
2. The outgoing connection edges for each channel parameter reached by a `!`

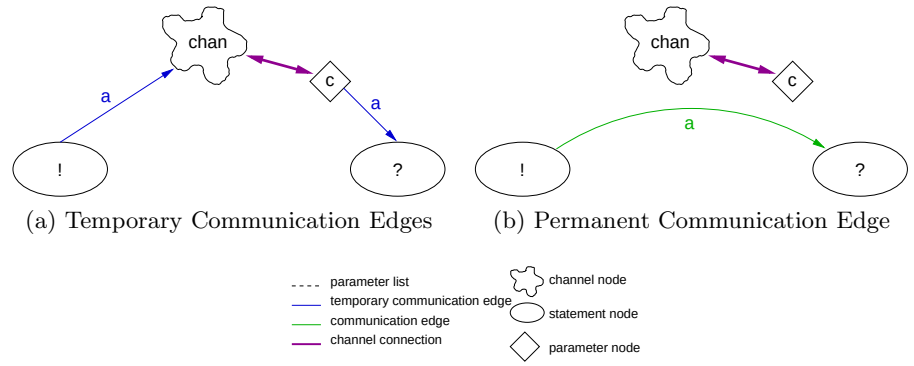


Figure 17: A ! communicating directly via a channel with a ? communicating via a parameter.

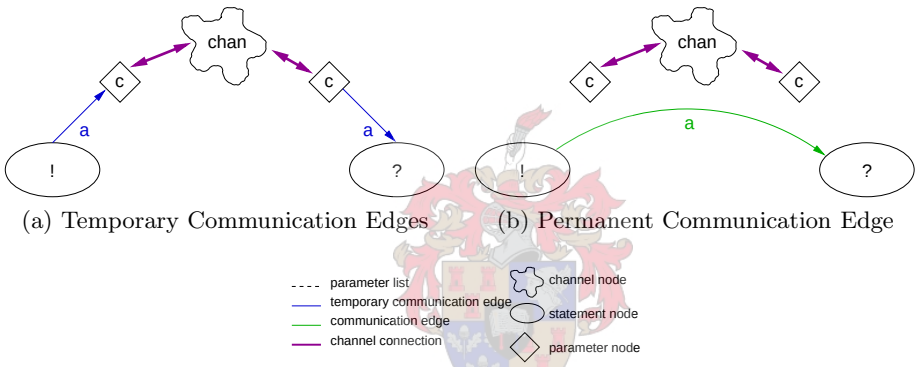


Figure 18: A ! and a ? communicating via parameters.

3. The outgoing connection and communication edges of a channel reached by either a channel parameter or a `!`

4. The outgoing communication edges from a channel parameter reached by a channel

Having explored the edges to find paths, `!` and `?` statements with common alphabet symbols are matched, however `!` and `?` statements belonging to the same process are only matched if there are multiple instances of the process present. The compiler calculates the number of times a call or `CREATE` to a specific process is made and sets a Boolean in the node representing the start of the process in the AST to indicate if it is encountered once or multiple times. It does this in a conservative fashion, assuming multiple calls if it is not possible to determine accurately. For instance, if a `CREATE` is situated within a `WHILE` loop, it is not known if the loop executes 0, 1 or many times,

so many is assumed.

3.4.2.2 Polling Communication

In Section 3.4.2.1, possible matches for communication statements within the body of a `WHEN` will have been found, but must still be confirmed. For this confirmation to take place, it must be checked that their potential communication partners occur after statements that could have enabled the `WHEN`.

Given a `WHEN` statement w, its matching partner v and a communication statement s within the statement block of w, we identify valid partners as follows: Find the first statement d that is a successor of v such that d and s use matching symbols. If d was identified as a possible partner for s when matching non-polling communication, then d may be confirmed as a permanent communication partner of s and a communication edge (d, s) is created within the CCDG.

3.5 Example

In this section the construction of a CCDG will be reviewed with the aid of an example. The source code for the example can be found in Listing 3.6 and the completed CCDG in Figure 20.

The program begins by initializing a global channel (`Chan`) with the `NEW` statement (line 35). Process `P1` is then instantiated with the `CREATE` statement (line 36) and executes in parallel with its parent, (the main program thread). Process `P1` receives `Chan` as a parameter and in turn instantiates `P0`, passing `chan`, an alias of `Chan`, as parameter (line 27). `P1` then attempts a number of communication statements (lines 28–31).

Communication in `LF` is synchronous so each of these statements will have to be completed before continuing on to the next. Process `P0` contains a `WHILE` loop (line 13) with a `SELECT` statement (line 14) in its body. The `SELECT` has two polling guards (lines 15 and 17). The first accepts a message with alphabet symbol `a` and data value `50` and the second a message with alphabet symbol `b` and data value `60`. If either one of the guards evaluate to TRUE, the body of the enabled `WHEN` will execute. In this case, both send a message with alphabet symbol `c` containing different data values (lines 16

Listing 3.6: Example.

```
1  MODULE Example;
2
3  TYPE
4    ChannelDesc = [a(LONGINT), b(LONGINT), c(LONGINT, LONGINT)];
5
6  VAR
7    Chan : ChannelDesc;
8
9  PROCESS P0(chan: ChannelDesc);
10 VAR
11   x: LONGINT;
12 BEGIN
13   WHILE TRUE DO
14     SELECT
15     WHEN chan ? a(x) & x = 50 THEN
16       chan ! c(30, 20)
17     WHEN chan ? b(x) & x = 60 THEN
18       chan ! c(30, 30)
19     END
20   END
21 END P0;
22
23 PROCESS P1(chan: ChannelDesc);
24 VAR
25   x, y: LONGINT;
26 BEGIN
27   CREATE P0(chan);
28   chan ! a(50);
29   chan ? c(x, y);
30   chan ! b(60);
31   chan ? c(x, y)
32 END P1;
33
34 BEGIN
35   NEW(Chan);
36   CREATE P1(Chan)
37 END Example.
```

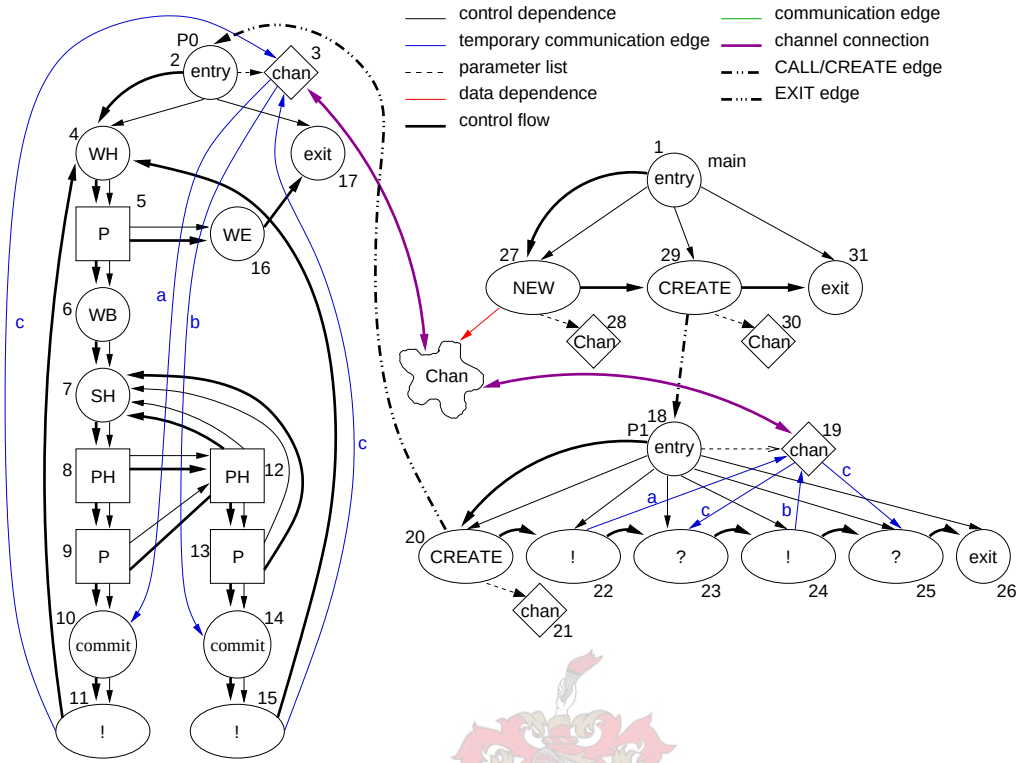


Figure 19: CCDG of the example in Listing 3.6 with temporary communication edges.

and 18).

Note that the nodes in Figures 19 and 20 are numbered in the order in which they were created when building the CCDG. Figure 19 contains the CCDG with temporary communication edges and Figure 20 the one with permanent communication edges.

Temporary communication edges of polling `?`s In Figure 19, node 8 represents the polling `?` of the `WHEN` in line 15. Node 9 represents the additional predicate and node 10 the region that controls the body of the `WHEN`. Note that the temporary communication edge is between nodes 3 and 10, and not nodes 3 and 8. During execution, the communication will only commit if a possible partner for the `?` exists and if the conditions of the additional predicate are met. Therefore, even though node 8 represents the polling `?`, the commit only takes place after the predicates are satisfied, so the region is used to model the point of communication. The same is true for polling `!`s.

Channel connections Connections between channels and parameters are established

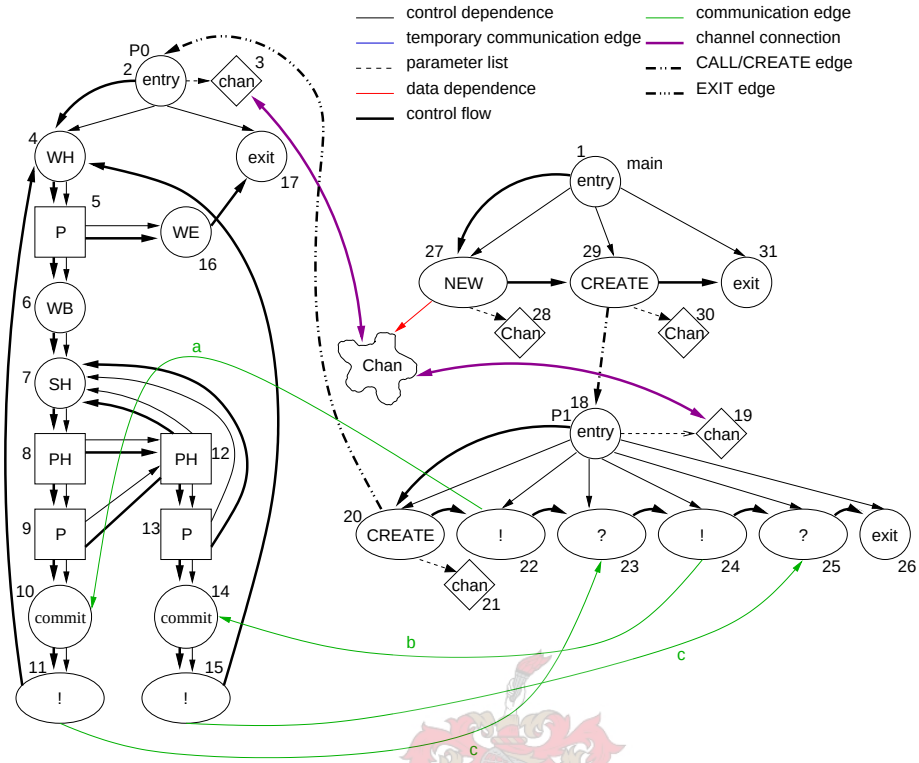


Figure 20: CCDG of the example in Listing 3.6 with resolved communication edges.

when `CREATE` or call statements are constructed. If the channel being passed as parameter is not a parameter itself, we can construct the channel connection edge immediately, given that the node representing the channel has been created. This is the case when building nodes 29 and 30 in Figure 19. They represent the `CREATE` statement instantiating process `P1`. The channel connection edge is created between the formal parameter `chan` (node 19) of `P1` and the global channel `Chan`. In the case where `P0` is instantiated, the channel being passed is a parameter. It is not known at this point if all the channel connections of that parameter has been created, so an entry is made into the list of unresolved channel connections of node 19 and node 3. When the whole CCDG has been constructed, this list will be traversed and all the connections of node 19 will be transferred to node 3.

Matching communication partners In Figure 20 the communication partners have been found and permanent communication edges created between the nodes representing them. The edge $(22, 10)$ will have been calculated as follows: by traversing edges $(22, 19)$, $(19, \texttt{Chan})$, $(\texttt{Chan}, 3)$ and $(3, 10)$ it is found that these two nodes

share a channel and they use matching alphabet symbols, therefore we add a permanent communication edge. The same process is followed in resolving the permanent communication edge $(24, 14)$.

When finding all the possible `?`s that could match up with node 11, both node 23 and node 25 are identified as possibilities. The following paths are traversed to establish this: $(11, 3)$, $(3, \texttt{Chan})$, $(\texttt{Chan}, 19)$, $(19, 23)$ and $(19, 25)$. However, node 11 resides within the control region of a `WHEN` clause, so the matching of a communication partner for it requires further scrutiny. Node 11 may only be matched with a communication statement that follows a statement that could have enabled the `WHEN`. It is found that node 23 is the first statement following node 22 (which is the matching communication partner for node 10) that can match with node 11, and so the permanent communication edge $(11, 23)$ is established. The partners of node 15 are resolved in a similar fashion.

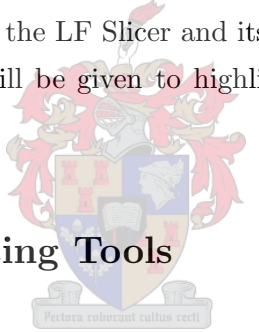
3.6 Summary

The *communicating concurrent dependence graph* (CCDG) was introduced in this chapter. It is based on a PDG and used to calculate slices of programs based on a specific criterion. The CCDG in conjunction with the AST produced by the compiler is used to derive the source code for such slices. This is discussed in Chapter 4. Extracts of the algorithm for the building of the different structures that make up CCDG accompany each explanation, as well as an example.

Chapter 4

The LF Slicer

The LF Slicer produces static backward slices based on a criterion consisting of a specified line of code and a variable used or defined at that point. In this chapter the slicing algorithm, the interface of the LF Slicer and its design and implementation will be discussed. Some examples will be given to highlight specific aspects deemed interesting.



4.1 An Overview of Slicing Tools

A number of slicing tools have been developed to address specific software engineering tasks. Most of these tools rely on graph reachability and are capable of computing interprocedural or intraprocedural slices.

The *Oberon Slicing Tool* (OST) was developed by Steindl to slice Oberon-2 programs [32] without any restrictions being placed on the features of the language one can use. The tool computes static slices interprocedurally, intraprocedurally and intermodularly and provides for both procedural and object-oriented programs. The slicing is expression-orientated as opposed to statement-orientated and uses user-feedback to restrict the effects of aliasing and dynamic binding. It stores information about computed slices for use when importing an already sliced module and provides multiple views of the same slice that are kept consistent.

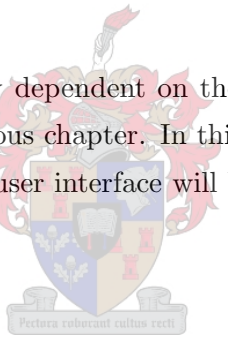
The *Wisconsin Program-Slicing Tool* (WPST)[1] was designed to facilitate program understanding and is capable of computing static forward and backward slices, as well as performing chopping (a topic not covered by this thesis) on programs written in C. It is also capable of displaying control and/or flow dependencies between program components.

Bandera is a tool set for model checking Java programs, part of which includes a slicer. The slicer is used to reduce Java programs to components that are relevant to a specific property. From these components a model is derived that is verified with either Spin or SMV [5].

Some commercial tools such as Menagerie (IBM[2]) and CodeSurfer (GrammaTech Inc[3]) are also available. CodeSurfer resulted from the commercialization of WPST.

4.2 Slicing in LF

The working of the LF Slicer is largely dependent on the correct construction of the CCDG that was discussed in the previous chapter. In this section, the algorithm that operates on the CCDG, as well as the user interface will be described.



4.2.1 The Algorithm

A slice is identified by marking nodes corresponding to statements during a traversal of the CCDG of a program. The algorithm used is based on a worklist approach. The worklist is initialized with the node that corresponds to the statement in the slicing criterion. Nodes are taken from the worklist one by one and marked as visited. All their relevant edges are explored and as unmarked nodes are encountered, they are placed in the worklist. During the execution of the algorithm the worklist grows and shrinks until it becomes empty. At this point the algorithm terminates and the source code for the slice can be reproduced from the AST using the marked nodes in the CCDG as guide.

Algorithm 9 takes a CCDG node and a set of variables as input. The variables are

[1] `http://www.cs.wisc.edu/wpis/html`
[2] `http://www.research.ibm.com/patv`
[3] `http://www.grammatech.com/products/codesurfer/index.html`

represented as AST nodes. The CCDG node, as mentioned above, represents the statement relevant to the slicing criterion as do the variables. The `worklist` is initialized with the `start` node (line 1), whereupon the algorithm initiates the loop that will terminate once the `worklist` is empty (line 2).

A node, `v`, is removed from the `worklist` and marked as visited (lines 3 and 4). Its control dependence predecessors, reaching definitions, communication partners and call sites are explored. All the previously unmarked control dependence predecessors are added to the `worklist`. Processing the reaching definitions is more complicated. If node `v` is the `start` node, then only those reaching definitions related to the variables in the slicing criterion are explored, otherwise all reaching definitions are explored. Next, a distinction is made between a reaching definition coming from an actual parameter, a formal parameter or any other node. Reaching definitions not coming from a parameter are added directly to the `worklist`. However, if the reaching definition is an actual parameter, the following applies:

- The actual parameter is marked as visited.
- The call/`CREATE` node to which the actual parameter belongs is added to the `worklist`.
- The call/`CREATE` node is added to the call stack to facilitate backtracking when descending into a process.
- The entry node of the instantiated process is marked as visited.
- The node representing the corresponding formal parameter is added to the `worklist`

In the case of the reaching definition being a formal parameter, the following steps apply:

- The formal parameter is marked as visited.
- If the call stack is not empty and the top node is a call to this process, do the following:
  - The last entry on the call stack is removed.
  - The node representing the corresponding actual parameter is added to the worklist.

Algorithm 9 The slicing algorithm

INPUT: start – Node in CCDG corresponding to statement in slicing criterion
Variables – Set of variables in slicing criterion

```
 1: worklist = worklist ∪ {start}
 2: while worklist ≠ ∅ do
 3:     v ← front(worklist)
 4:     mark v as visited
 5:     for all control dependence predecessors cd of v do
 6:         if cd not visited then
 7:             worklist ← worklist ∪ {cd}
 8:         end if
 9:     end for
10:     for all reaching definition rd of v do
11:         if (((v = start) & (variable of rd ∈ Variables)) OR (v ≠ start)) & (rd not visited) then
12:             if rd is an actual parameter then
13:                 mark rd as visited
14:                 worklist = worklist ∪ {the instantiating node to which rd belongs}
15:                 callstack = instantiating node + callstack
16:                 mark the entry node of the instantiated process as visited
17:                 add the corresponding formal parameter to the worklist
18:             else if rd is a formal parameter then
19:                 mark rd as visited
20:                 if (callstack is not empty) & (head of callstack is an instantiation of the entry node of rd)
                    then
21:                     lastcaller ← remove the front of the callstack
22:                     add the corresponding actual parameter of lastcaller to the worklist
23:                 else
24:                     for all call edges c of rd's entrynode do
25:                         add the corresponding actual parameter of c to the worklist
26:                     end for
27:                 end if
28:             else
29:                 worklist ← worklist ∪ {rd}
30:             end if
31:         end if
32:     end for
33:     if type(v) ∈ {'bang','hook','bangrelease','hookcommit'} then
34:         for all incoming communication c of v do
35:             if c not visited then
36:                 worklist ← worklist ∪ {c}
37:                 worklist ← worklist ∪ {instantiation node of channel used by c}
38:             end if
39:         end for
40:         for all outgoing communication c of v do
41:             if c not visited then
42:                 worklist ← worklist ∪ {c}
43:                 worklist ← worklist ∪ {instantiation node of channel used by c}
44:             end if
45:         end for
46:     end if
47:     for all call site c of v do
48:         if (c not visited) & ((v = entrynode of start) OR (type(c) = 'create')) then
49:             worklist ← worklist ∪ {c}
50:         end if
51:     end for
52: end while
```

Module	Lines	Executable (bytes)
LFSD	457	7870
LFSCCDG	2265	39159
LFSM	189	3927
LFSTG	321	8878
LFSTT	116	941
LFSG	558	14785
LFST	512	5518
Total	4418	81078

Table 2: Breakdown of implementation into modules.

- Otherwise all possible instantiating nodes of this process must be explored, using these steps:
  - Each corresponding actual parameter of all the instantiations of this process are added to the `worklist`.

If the node has communication partners, they are added to the `worklist`. Additionally, the node that represents the instantiation of the channel used in these exchanges will also be added to the `worklist`.

If the node being explored is the entry node of the process to which the `start` node belongs, all its call edges are followed and the corresponding instantiation nodes are added to the `worklist`. For all other entry nodes, only those edges that lead to `CREATE` nodes are added to the `worklist`.

4.2.2 Design and Implementation

The implementation of the LF Slicer is divided into 7 modules. Table 2 shows the number of lines making up each of these modules as well as the size (in bytes) of the executables.

LFSD This module contains the definitions of all the data structures used in the implementation of the CCDG, as well as some procedures to manipulate them. General definitions used by other modules are also contained within `LFSD`.

LFSCCDG All the procedures for building a CCDG can be found in this module. It has a main procedure that takes the AST produced by the LF Compiler as input

and traverses it to construct the CCDG. Many helper procedures are defined.

LFSM The slicing algorithm is implemented in this module. One main procedure is defined that takes a CCDG node as input and traverses the CCDG to mark all the nodes relevant to the slicing criterion.

LFSTG Procedures for manipulating gadgets are found here. They are used to make the interface interactive.

LFSTT This module provides procedures for identifying a collection of selected gadgets as well as their order of selection.

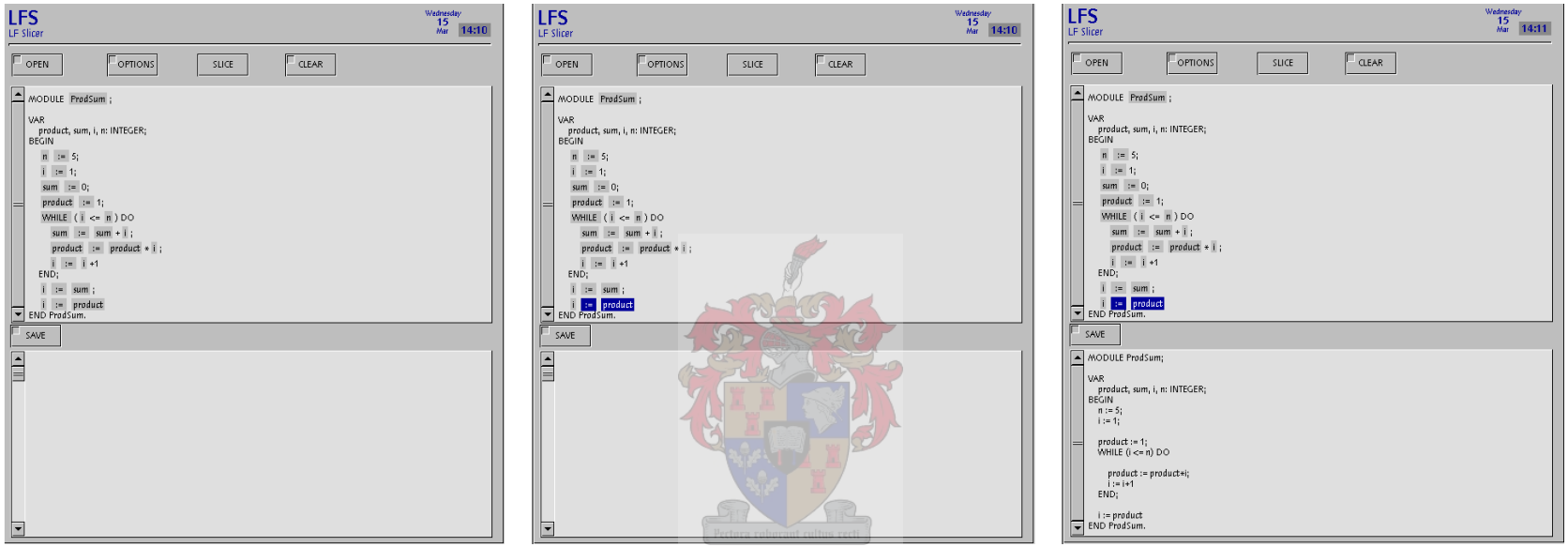
LFSG Source code generation from the AST is provided in this module. It can also relate the AST nodes back to their matching CCDG nodes to determine the inclusion of a program statement in a slice.

LFST This module provides the routines for the user interface and controls the loading, compiling, slicing and source code regeneration of programs.

4.2.3 The Interface

The main aim of the user interface is to be simple, easy to use and informative. Identifying the slicing criterion is the main concern. Hypertext, a specialized form of active text, was chosen. Active text provides an extension to traditional text by creating sensitive areas for the user to interact with. These sensitive areas may encapsulate a variety of information [23]. Traditionally, hypertext is used to link separate, but related pieces of information [23, 28]. In the case of the LF Slicer, parts of the source code will be linked to their corresponding nodes in the CCDG, which in turn are linked to the AST. In this way, it is easy to pass the relevant nodes on to the slicing algorithm.

The slicing tool generates source code with links to nodes by traversing the AST produced by the LF Compiler. The links are created by embedding visual objects into the text. These objects are based on Gadgets, a component framework of Oberon [20]. The components of the Gadget framework are also called gadgets and represent persistent objects that exist within the run-time environment of Oberon. The gadgets embedded within the source code are visually similar to normal text and are distinguished based on their bold type face and a grey block around the text. Figure 21a contains a view of the interface with a loaded source file containing active text. The grey blocks become blue when selected as shown in Figure 21b.



(a) LF Slicer Interface with a source file loaded into the window.

(b) LF Slicer Interface with slicing criterion marked on source file.

(c) LF Slicer Interface showing resulting slice.

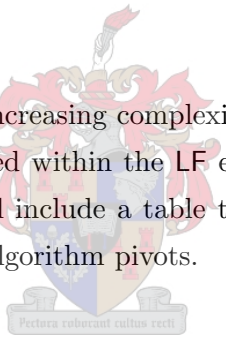
Figure 21: The LF Slicer Interface

worklist	v	CD	RD	Comms Partner	Call/Create	Marked
{14}	-	-	-	-	-	-
∅	14	1	5, 10	-	-	14
{1, 5}	10	8	3, 11, 5	-	-	14, 10
{1, 3, 5, 8}	11	8	3, 11	-	-	14, 10, 11
{1, 5, 8}	3	1	-	-	-	14, 10, 11, 3
{5, 8}	1	-	-	-	-	14, 10, 11, 3, 1
{5}	8	7	-	-	-	14, 10, 11, 3, 1, 8
{5}	7	6	2, 3, 11	-	-	14, 10, 11, 3, 1, 8, 7
{5, 6}	2	1	-	-	-	14, 10, 11, 3, 1, 8, 7, 2
{5}	6	1	-	-	-	14, 10, 11, 3, 1, 8, 7, 2, 6
∅	5	1	-	-	-	14, 10, 11, 3, 1, 8, 7, 2, 6, 5

Table 3: Worklist of execution of slicing algorithm to produce the slice in Listing 4.2.

4.2.4 Examples

In the following sections examples of increasing complexity will be discussed with the aim to show how slicing is accomplished within the LF environment and to illustrate the algorithm used. Each example will include a table that states the progression of the worklist around which the slicing algorithm pivots.



4.2.4.1 Traditional Product/Sum Example

The product/sum example is one that is found most often in slicing literature. Listing 4.1 contains an implementation of this example in LF. This specific example calculates the sum and product of the numbers between 1 and 5. It is a useful example because it clearly illustrates the principle of slicing. Figure 22 shows the generated CCDG. The nodes are numbered in the order in which they are created during construction.

A slice is taken on the criterion (15, `product`). The statement in the criterion is represented by node 14. This will form the `start` node for the slicing algorithm and will be the first node in the worklist as shown in Table 3. By following the algorithm in 9, node 14 is removed from the worklist and marked as visited.

Listing 4.1: Traditional Product/Sum example.

```
1  MODULE ProdSum;
2  VAR
3    product, sum, n, i : INTEGER;
4  BEGIN
5    n := 5;
6    i := 1;
7    sum := 0;
8    product := 1;
9    WHILE (n >= i) DO
10     sum := sum + i;
11     product := product * i;
12     i := i + 1;
13   END;
14   i := sum;
15   i := product;
16 END ProdSum.
```

Node 1 is identified as a control dependence predecessor and nodes 5 and 10 are identified as reaching definitions. As these nodes are not marked yet, they are placed in the worklist. Duplicates are not necessary in the worklist because one exploration of a node is sufficient to identify all its unmarked predecessors.

Node 10 is the next one that will be removed from the worklist, marked and have it's predecessors explored. It has node 8 as a control dependence predecessor and nodes 3, 11 and 5 as reaching definitions. As 5 is already in the worklist, only nodes 3 and 8 are added. This process is repeated for each node in the worklist until it becomes empty. The whole progression is given in Table 3.

Figure 23 shows the CCDG with the marked nodes coloured in grey. These nodes correspond to the statements that will be retained when reproducing the source code for the slice, as shown in Listing 4.2.

4.2.4.2 Product/Sum with a `SELECT`

In this example, the traditional product/sum example is modified to use a process that takes requests for sum or product calculations via a message that is passed on a channel. A `SELECT` is used to distinguish between the two messages. The appropriate answer is calculated and sent back via another channel. Figure 24 shows the completed CCDG for this program and Figure 25 shows the CCDG with the nodes relevant to the slice marked in grey. As for the previous example, Table 4 contains the progression of the worklist as the slicing algorithm performs on the source code in Listing 4.3.

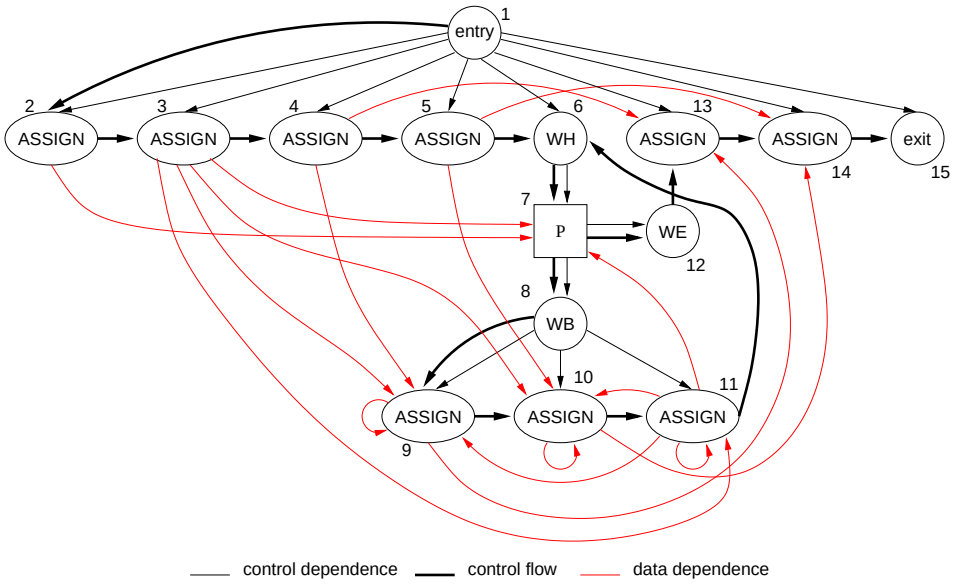


Figure 22: CCDG of the traditional Product/Sum example in Listing 4.1.

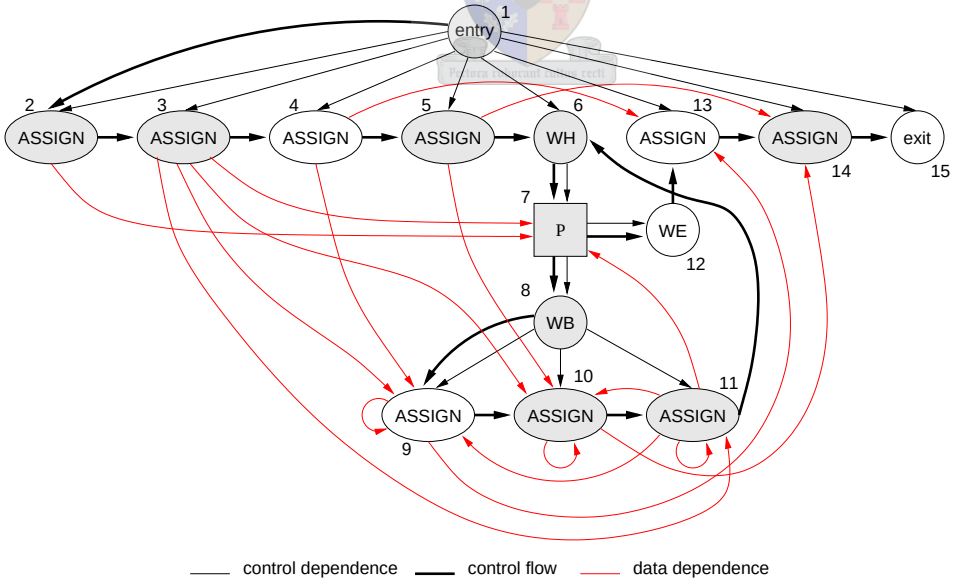


Figure 23: CCDG of the traditional Product/Sum example in Listing sliced on (15, `product`).

Listing 4.2: Slice of traditional Product/Sum example on (15, product).

```
1  MODULE ProdSum;
2  VAR
3    product, sum, n, i : INTEGER;
4  BEGIN
5    n := 5;
6    i := 1;
7
8    product := 1;
9    WHILE (n >= i) DO
10
11     product := product * i;
12     i := i + 1;
13   END;
14
15   i := product;
16 END ProdSum.
```

Table 4: Worklist of execution of slicing algorithm to produce slice in Listing 4.4.

worklist	v	CD	RD	Comms Partner	Call	Marked
{40}	-	-	-	-	-	-
∅	40	33	39	-	-	40
{33}	39	33	-	30, 44	-	40, 39
{30, 33}	44	1	-	-	-	40, 39, 44
{30, 33}	1	-	-	-	-	40, 39, 44, 1
{33}	30	21	23, 27	39, 44	-	40, 39, 44, 1, 30
{21, 23, 33}	27	26	22, 23, 27, 28	-	-	40, 39, 44, 1, 30, 27
{21, 22, 23, 26, 33}	28	26	22, 28	-	-	40, 39, 44, 1, 30, 27, 28
{21, 23, 26, 33}	22	21	-	-	-	40, 39, 44, 1, 30, 27, 28, 22
{23, 26, 33}	21	20	-	38, 42	-	40, 39, 44, 1, 30, 27, 28, 22, 21
{20, 23, 26,33, 38}	42	1	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42
{20, 23, 26, 33}	38	33	-	21, 42	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38
{20, 23, 26}	33	-	-	-	49	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33
{20, 23, 26}	49	1	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49
{23, 26}	20	9	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20
{23, 26}	9	8	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9
{23, 26}	8	7	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8

Continued on next page

worklist	v	CD	RD	Comms Partner	Call	Marked
{23, 26}	7	6	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7
{23, 26}	6	5	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7, 6
{23, 26}	5	2	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7, 6, 5
{23, 26}	2	-	-	-	46	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7, 6, 5, 2
{23, 26}	46	1	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7, 6, 5, 2, 46
{23}	26	25	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7, 6, 5, 2, 46, 26
{23}	25	24	21, 22, 28	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7, 6, 5, 2, 46, 26, 25
{23}	24	21	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7, 6, 5, 2, 46, 26, 25, 24
∅	23	21	-	-	-	40, 39, 44, 1, 30, 27, 28, 22, 21, 42, 38, 33, 49, 20, 9, 8, 7, 6, 5, 2, 46, 26, 25, 24, 23

worklist	v	CD	RD	Comms Partner	Call/Create	Marked
{16}	-	-	-	-	-	-
∅	16	10	14	-	-	16
{10}	14	-	-	-	-	16, 14, 5
{10, 13}	6	-	8	-	-	16, 14, 5, 6
{10, 13}	8	5	7	-	-	16, 14, 5, 6, 8, 7
{10, 13}	15	-	12	-	-	16, 14, 5, 6, 8, 7, 15
{10, 13}	12	10	-	-	-	16, 14, 5, 6, 8, 7, 15, 12
{13}	10	-	-	-	-	16, 14, 5, 6, 8, 7, 15, 12, 10
{13}	20	1	-	-	-	16, 14, 5, 6, 8, 7, 15, 12, 10, 20
{13}	1	-	-	-	-	16, 14, 5, 6, 8, 7, 15, 12, 10, 20, 1
∅	13	10	-	-	-	16, 14, 5, 6, 8, 7, 15, 12, 10, 20, 1, 13

Table 5: Worklist of execution of slicing algorithm to produce slice in Listing 4.6.

4.2.4.3 Parameters and Calls

Listing 4.5 contains the source code for a program used to illustrate the concept of parameters and how they can be sliced in LF. This example does not do anything meaningful and is purely for illustrative purposes. The main process instantiates two processes, `P0` and `P1`, with the aid of LF's `CREATE` statement. Process `P0` updates the value of the global variable `X`. Process `P2` initializes its two local variables `x` and `y`, calls process `P1` with `x` and `y` as parameters and upon the return from `P1` updates the value of `X` with the sum of `x`, `y` and `X`. Process `P1` takes a call-by-reference and a call-by-value parameter as input and assigns the value of `b*2` to `a`.

Listing 4.6 contains the slice of the example in Listing 4.5 when using (22, `x`) as criterion. Process `P0` is sliced away completely as the value of the global variable is not relevant to the value of `x` at that point in the program. The initialization of `x` in `P2` is also taken away as `x` is redefined by the reference parameter in process `P1`. The instantiation of process `P0` as well as the initialization of the global variable is also removed from the main process body. Table 5 contains the trace of the worklist when executing the slicer to produce this slice.

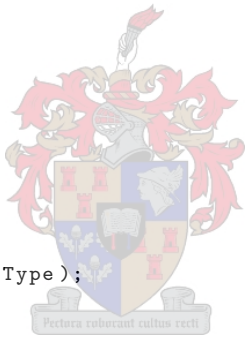
Listing 4.7 shows the same example sliced on (22, `y`) and Listing 4.8 the slice using

Listing 4.3: Product/Sum Example with a SELECT.

```
1  MODULE ProdSumSelect;
2  TYPE
3    ChannelType = [sum(LONGINT), product(LONGINT), answer(LONGINT)];
4
5  PROCESS Compute(in, out : ChannelType);
6  VAR
7    i, n, sum, prod : LONGINT;
8  BEGIN
9    WHILE TRUE DO
10     SELECT
11     WHEN in ? sum(n) THEN
12       i = 1;
13       sum = 0;
14       WHILE i <= n DO
15         sum := sum + i;
16         i := i + 1;
17       END;
18       out ! answer(sum);
19     WHEN in ? product(n) THEN
20       i = 1;
21       prod = 1;
22       WHILE i <= n DO
23         prod := prod * i;
24         i := i + 1;
25       END;
26       out ! answer(prod);
27     END;
28   END;
29 END Compute;
30
31 PROCESS Client(in, out : ChannelType);
32 VAR
33   x, sum, prod : LONGINT;
34 BEGIN
35   out ! sum(10);
36   in ? answer(sum);
37   out ! product(10);
38   in ? answer(prod);
39   x := sum + prod;
40 END Client;
41
42 VAR
43   ask, ans : ChannelType;
44 BEGIN
45   NEW(ask);
46   NEW(ans);
47   CREATE Compute(ask, ans);
48   CREATE Client(ans, ask);
49 END ProdSumSelect.
```

Listing 4.4: Slice on Product/Sum Example with a SELECT on (39, prod).

```
1  MODULE ProdSumSelect;
2  TYPE
3    ChannelType = [sum(LONGINT), product(LONGINT), answer(LONGINT)];
4
5  PROCESS Compute(in, out : ChannelType);
6  VAR
7    i, n, sum, prod : LONGINT;
8  BEGIN
9    WHILE TRUE DO
10     SELECT
11
12
13     WHEN in ? product(n) THEN
14       i = 1;
15       prod = 1;
16       WHILE i <= n DO
17         prod := prod * i;
18         i := i + 1;
19       END;
20       out ! answer(prod);
21     END;
22   END;
23 END Compute;
24
25 PROCESS Client(in, out : ChannelType);
26 VAR
27   x, sum, prod : LONGINT;
28 BEGIN
29
30
31   out ! product(10);
32   in ? answer(prod);
33   x := sum + prod;
34 END Client;
35
36 VAR
37   ask, ans : ChannelType;
38 BEGIN
39   NEW(ask);
40   NEW(ans);
41   CREATE Compute(ask, ans);
42   CREATE Client(ans, ask);
43 END ProdSumSelect.
```



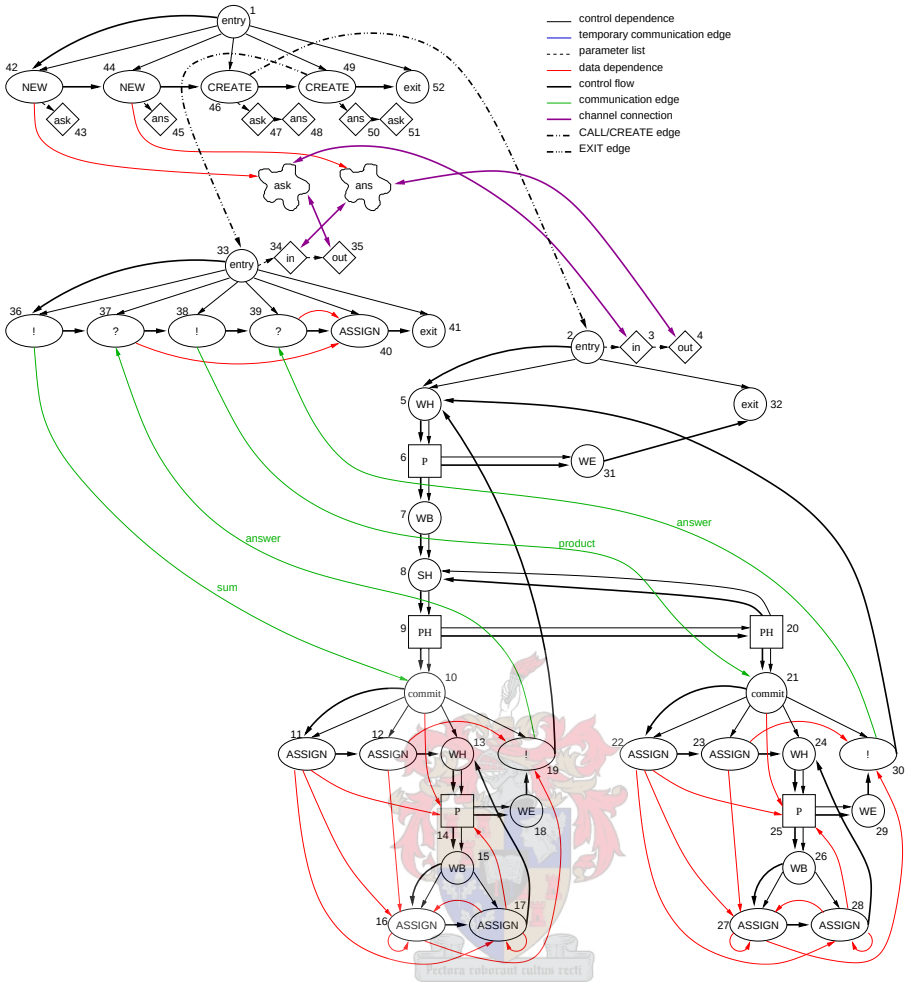


Figure 24: CCDG of the Product/Sum example with a `SELECT` in Listing 4.3.

criterion (22, `X`).

4.3 Summary

The slicing algorithm was presented in this chapter. Its working was explained and illustrated with a number of examples. The implementation of the algorithm was discussed in conjunction with the user interface. In Chapter 5 the goals are restated and an evaluation of the project is presented along with suggestions for improvements.

Listing 4.5: Parameters.

```
1  MODULE Parameter;
2  VAR
3    X : LONGINT;
4
5  PROCESS P0;
6  BEGIN
7    X := 20;
8  END P0;
9
10 PROCESS P1(VAR a : LONGINT; b : LONGINT);
11 BEGIN
12   a := b * 2;
13 END P1;
14
15 PROCESS P2;
16 VAR
17   x, y : LONGINT;
18 BEGIN
19   x := 5;
20   y := 5;
21   P1(x, y);
22   X := x + y + X;
23 END P2;
24
25 BEGIN
26   X := 5;
27   CREATE P0;
28   CREATE P2;
29 END Parameter;
```

Listing 4.6: Parameters example sliced on (22, x).

```
1  MODULE Parameter;
2  VAR
3    X : LONGINT;
4
5
6  PROCESS P1(VAR a : LONGINT; b : LONGINT);
7  BEGIN
8    a := b * 2;
9  END P1;
10
11 PROCESS P2;
12 VAR
13   x, y : LONGINT;
14 BEGIN
15
16   y := 5;
17   P1(x, y);
18   X := x + y + X;
19 END P2;
20
21 BEGIN
22
23
24   CREATE P2;
25 END Parameter;
```

Listing 4.7: Parameters example sliced on (22, y).

```
1  MODULE Parameter;
2  VAR
3    X : LONGINT;
4
5
6
7  PROCESS P2;
8  VAR
9    x, y : LONGINT;
10 BEGIN
11
12   y := 5;
13
14   X := x + y + X;
15 END P2;
16
17 BEGIN
18
19
20   CREATE P2;
21 END Parameter;
```

Listing 4.8: Parameters example sliced on (22, X).

```
1  MODULE Parameter;
2  VAR
3    X : LONGINT;
4
5  PROCESS P0;
6  BEGIN
7    X := 20;
8  END P0;
9
10 PROCESS P1(VAR a : LONGINT; b : LONGINT);
11 BEGIN
12   a := b * 2;
13 END P1;
14
15 PROCESS P2;
16 VAR
17   x, y : LONGINT;
18 BEGIN
19
20   y := 5;
21   P1(x, y);
22   X := x + y + X;
23 END P2;
24
25 BEGIN
26   X := 5;
27   CREATE P0;
28   CREATE P2;
29 END Parameter;
```

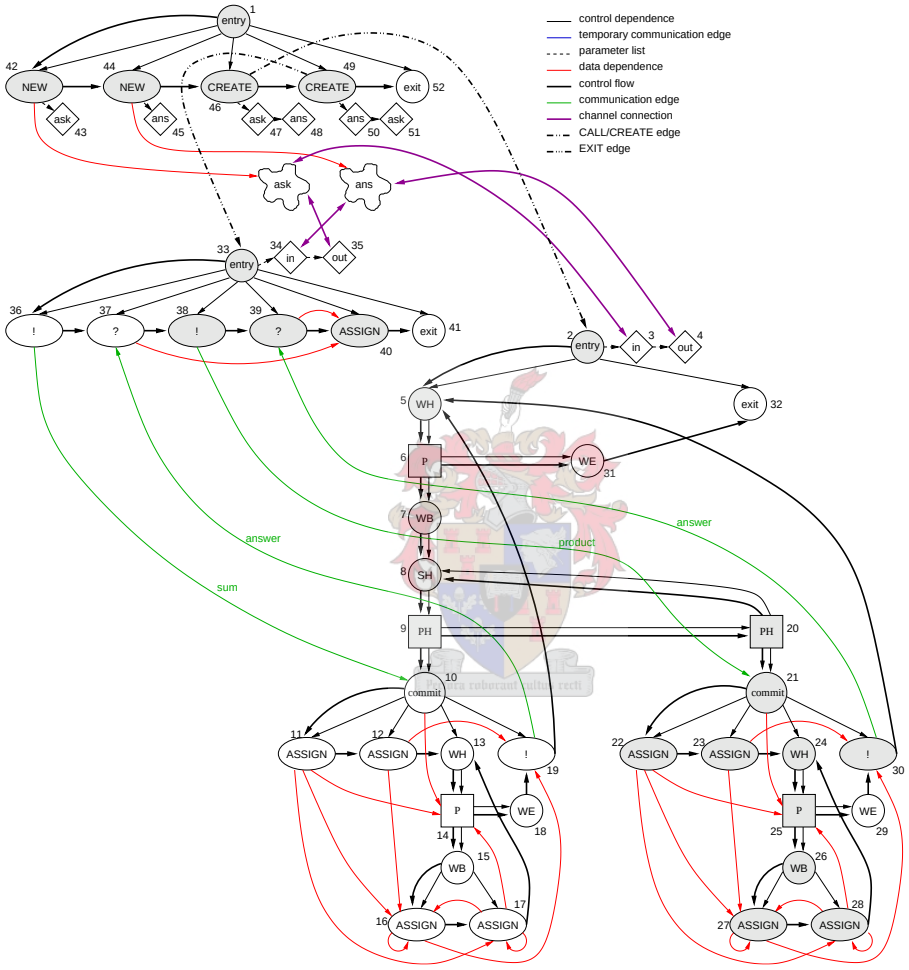


Figure 25: CCDG of the Product/Sum example with a `SELECT` in Listing 4.3 sliced on (39, `prod`).

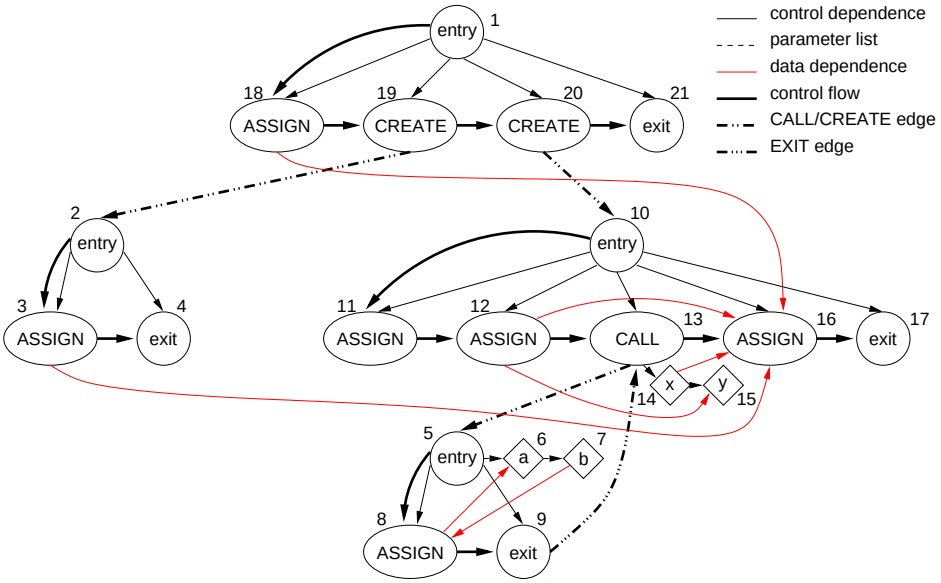


Figure 26: CCDG of the Parameter example in Listing 4.5.

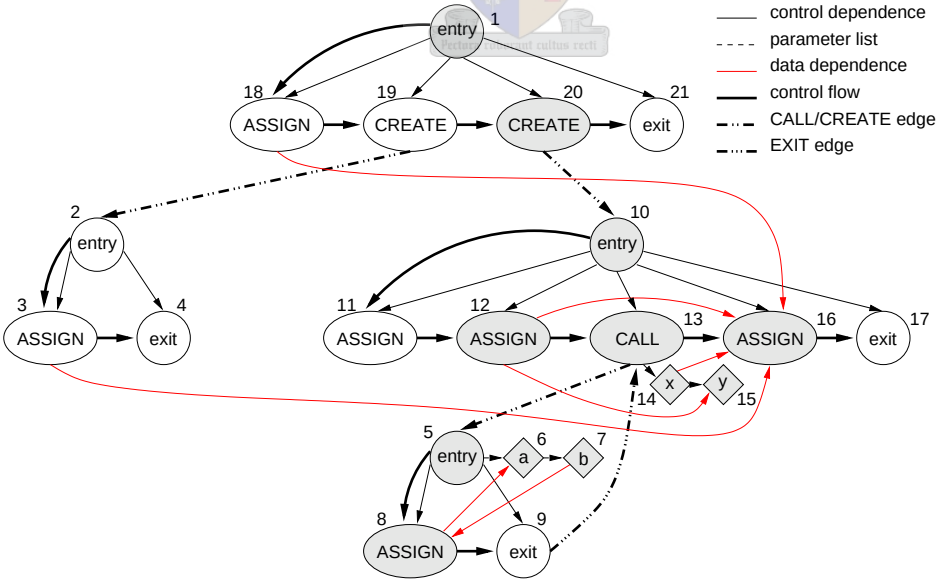


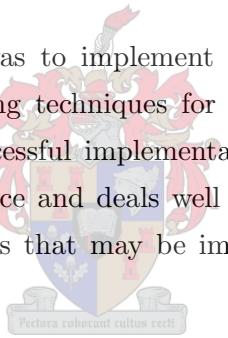
Figure 27: CCDG of the Parameter example in Listing 4.5 sliced on (22, x).

Chapter 5

Conclusion

5.1 The Goals

The goal, as stated in Section 1.1, was to implement a slicer for an experimental concurrent language called LF. Existing techniques for slicing procedural languages were adapted and resulted in the successful implementation of a program slicer. A working slicer for LF is now in existence and deals well with the concurrency model of LF, although there are some aspects that may be improved upon or investigated further.



5.2 Evaluation

The accuracy of the context-sensitivity analysis of communication statements inside `SELECT` statements can be improved. Currently it is assumed that matching communication statements for those in the body of a `WHEN` will always be in the same process as the communication statement that enabled the `WHEN`.

Another point to highlight is the inaccuracy of the data-flow analysis for global variables. This is discussed in section 5.3.1.

Table 6 contains a list of all the components of a CCDG and their respective sizes. Tables 7, 8 and 9 contain a list of all the examples given throughout the thesis along with their corresponding CCDG sizes. A breakdown of the components is also given for each example.

Component	Size
CCDGNode	116
Control Dependence Edge	12
Control Flow Edge	8
Data Dependence Edge	12
Parameter	8
Call/Create Edge	12
Exit Edge	12
Communication Edge	52
Channel Connection Edge	12
Channel Node	84

Table 6: Size (in bytes) of components making up CCDG.

Example	Total Graph Size
Figure 6	1912
Figure 7	1332
Figure 8	1924
Figure 11	3092
Figure 14	6588
Figure 20	5820
Figure 22	2520
Figure 24	9408
Figure 26	3164

Table 7: Total graph size (in bytes) for each example.

The sizes of the graphs seem to be quite manageable. A major contributor is the size of the communication edges. These could raise some scalability issues, but a larger example (source not given here) of a replicated file server consisting of 286 lines of code produces a graph of less than 67KB. This is promising for the scalability of the graph structure.

5.3 Future Work

During the course of this thesis some points that require further exploration were touched upon and are discussed here in greater detail.

Example	Nodes	Control Dependence	Control Flow	Data Dependence
Figure 6	1392	264	208	48
Figure 7	928	216	128	60
Figure 8	1508	168	112	48
Figure 11	1972	264	176	36
Figure 14	3828	720	528	96
Figure 20	3596	624	448	12
Figure 22	1740	336	240	240
Figure 24	6032	960	704	312
Figure 26	2436	312	208	72

Table 8: Breakdown of CCDG Components (sizes in bytes).

Example	Parameter	Call/Create	Exit	Communication	Connection	Channel
Figure 6	-	-	-	-	-	-
Figure 7	-	-	-	-	-	-
Figure 8	64	12	12	-	-	-
Figure 11	48	24	24	416	48	48
Figure 14	48	12	12	1092	-	252
Figure 20	80	24	24	832	96	84
Figure 22	-	-	-	-	-	-
Figure 24	160	24	24	832	192	168
Figure 26	64	36	36	-	-	-

Table 9: Breakdown of CCDG Components (sizes in bytes).

5.3.1 Global Variables

Currently the slicer deals with global variables in a context-insensitive way, assuming that all definitions of global variables can reach all uses of global variables at any point within a program. This is not necessarily true in all instances. The concurrent nature of LF complicates the analysis of global variables as scheduling is unpredictable, making it difficult to determine which processes are active at a given point and which statements may have executed.

5.3.1.1 MHP

The may-happen-in-parallel (MHP) algorithm in [26] and [27] was adapted for LF in an attempt to investigate its application to the slicer and specifically to global data flow

analysis. The algorithm works on a graph where nodes represent statements. For each node, it identifies in a conservative fashion, all other nodes that could happen in parallel with it. The goal was to use the MHP information to identify all the statements that have been executed before a given statement is reached. For example, in the following code fragment, if statement 2 executes, statements 0 and 1 must have executed already because the statements inside a process are executed sequentially.

```
...
PROCESS P0;
VAR x, y : LONGINT;
BEGIN
  x := 5;     (* stmt 0 *)
  y := 6;     (* stmt 1 *)
  x := x + y; (* stmt 2 *)
END P0;
...
```

The must-have-happened (MHH) information can then be used in conjunction with the MHP sets to determine which global definitions will reach a specific use. MHH information for each process is calculated independently by adding all previous statements to the MHH set of a current statement. For example, in the aforementioned code fragment, the MHH set for statement 1 will be statements $\{0, 1\}$, and for statement 2 it will be statements $\{0, 1, 2\}$. The current statement is included in its own MHH set.

When a statement uses a global variable, a more complete MHH set is calculated as the union of the MHH sets of all the statements in the statement's MHP set. If the definition that could possibly reach this use is in a different process, the MHH set of all statements that could have instantiated this process are included as well. It is then verified whether the possible reaching definition is in this completed MHH set. If so, the reaching definition is included. This solution however does not work when LF's synchronous communication needs to be taken into account.

Table 10 shows the MHP and MHH sets for each node in Figure 28. This figure shows the CCDG as it should be if reaching definition information for global variables could be calculated accurately using the method previously described. The example in Listing 5.1 contains three processes and a main body. The three processes execute in parallel and contain endless loops, passing each other a signal indicating it is the next

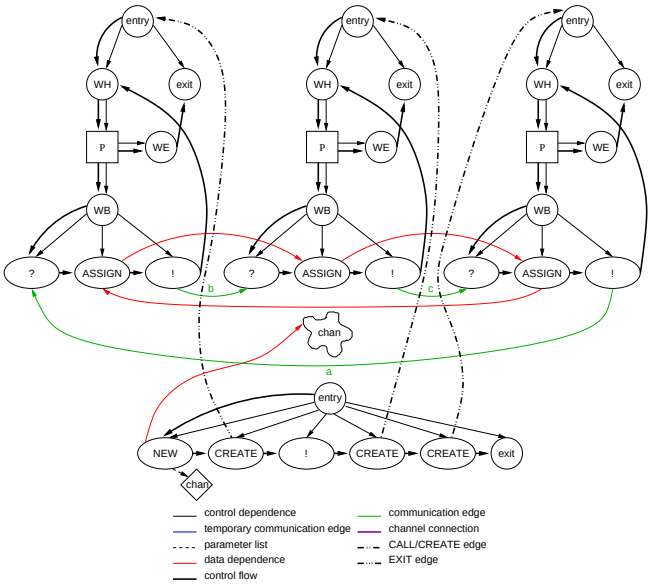


Figure 28: The CCDG of the Lock Step example in Listing 5.1.

ones turn to update the global variable.

In line 21, process `P1` is updating the global variable by adding 23 to it. The possible definitions that can reach this statement are found in lines 12 and 30. When looking closely at the synchronization of the processes via the passing of signals, only the definition in process `P0` should reach the use in process `P1`. But the MHH set of node 16, representing line 21, includes neither node 7 nor node 25. This is because the MHP set of node 16 is empty and therefore none of the nodes in the other processes are included in the MHH set. The reason for the MHP set being empty requires further investigation.

5.3.2 Intermodular Slicing

A useful functionality to have would be that of intermodular slicing. This would require extending the CCDG to accommodate multiple modules. At first thought, this seems fairly straightforward to implement, simply handling processes from other modules in the same way as processes in one module are handled. This would be the reasonable starting point, given that one has access to the source code of the imported module.

Some questions that need further investigation are:

Listing 5.1: Lock Step Example.

```
1  MODULE Lockstep;
2  TYPE
3    chanType = [a, b, c];
4  VAR
5    chan : chanType;
6    x : LONGINT;
7
8  PROCESS P0;
9  BEGIN
10   WHILE TRUE DO
11     chan ? a;
12     x := x + 2;
13     chan ! b;
14   END;
15 END P0;
16
17 PROCESS P1;
18 BEGIN
19   WHILE TRUE DO
20     chan ? b;
21     x := x + 23;
22     chan ! c;
23   END;
24 END P1;
25
26 PROCESS P2;
27 BEGIN
28   WHILE TRUE DO
29     chan ? c;
30     x := x + 32;
31     chan ! a;
32   END;
33 END P2;
34
35 BEGIN
36   NEW(chan);
37   CREATE P0;
38   chan ! a;
39   CREATE P1;
40   CREATE P2;
41 END Lockstep.
```

Node	MHP	MHH
{1}	∅	{1}
{29}	∅	{1, 29}
{31}	∅	{1, 29, 31}
{32}	∅	{1, 29, 31, 32}
{33}	∅	{1, 29, 31, 32, 33}
{34}	{12, 13, 18, 19, 14, 11}	{1, 29, 31, 32, 33, 34}
{35}	{21, 22, 27, 28, 23, 11, 14, 19, 18, 13, 12, 20}	{1, 29, 31, 32, 33, 34, 35}
{2}	∅	{2}
{3}	∅	{2, 3, 4, 5, 6, 7, 8}
{4}	∅	{2, 3, 4, 5, 6, 7, 8}
{5}	∅	{2, 3, 4, 5, 6, 7, 8}
{6}	∅	{2, 3, 4, 5, 6, 7, 8}
{7}	∅	{2, 3, 4, 5, 6, 7, 8, 31, 29, 1}
{8}	∅	{2, 3, 4, 5, 6, 7, 8}
{9}	∅	{2, 3, 4, 5, 6, 7, 8, 9}
{10}	∅	{2, 3, 4, 5, 6, 7, 8, 9, 10}
{11}	{34, 20, 21, 22, 27, 28, 23, 35}	{11}
{12}	{35, 23, 28, 27, 22, 21, 20, 34}	{11, 12, 13, 14, 15, 16, 17}
{13}	{34, 20, 21, 22, 27, 28, 23, 35}	{11, 12, 13, 14, 15, 16, 17}
{14}	{35, 23, 28, 27, 22, 21, 20, 34}	{11, 12, 13, 14, 15, 16, 17}
{15}	∅	{11, 12, 13, 14, 15, 16, 17}
{16}	∅	{11, 12, 13, 14, 15, 16, 17, 33, 32, 31, 29, 1}
{17}	∅	{11, 12, 13, 14, 15, 16, 17}
{18}	{35, 23, 28, 27, 22, 21, 20, 34}	{11, 12, 13, 14, 15, 16, 17, 18}
{19}	{34, 20, 21, 22, 27, 28, 23, 35}	{11, 12, 13, 14, 15, 16, 17, 18, 19}
{20}	{11, 14, 19, 18, 13, 12, 35}	{20}
{21}	{14, 19, 18, 13, 12, 35, 11}	{20, 21, 22, 23, 24, 25, 26}
{22}	{35, 11, 14, 19, 18, 13, 12}	{20, 21, 22, 23, 24, 25, 26}
{23}	{35, 11, 14, 19, 18, 13, 12}	{20, 21, 22, 23, 24, 25, 26}
{24}	∅	{20, 21, 22, 23, 24, 25, 26}
{25}	∅	{20, 21, 22, 23, 24, 25, 26, 34, 33, 32, 31, 29, 1}
{26}	∅	{20, 21, 22, 23, 24, 25, 26}
{27}	{35, 11, 14, 19, 18, 13, 12}	{20, 21, 22, 23, 24, 25, 26, 27}
{28}	{35, 11, 14, 19, 18, 13, 12}	{20, 21, 22, 23, 24, 25, 26, 27, 28}

Table 10: MHP and MHH sets for example in Listing 5.1

- If only the object code of a module is available, how should calls to processes located inside the imported module be handled?
- How should global variables visible outside the scope of a module be handled, as opposed to global variables only visible within one module?

5.3.3 Support for Other Languages

The adaption of the CCDG for use with another language should be a feasible exercise. For any procedural language with call-by-value and call-by-reference parameters one should be able to use the CCDG as is, only making minor modifications to the interaction with the AST of the language.

For languages such as Ada, occam and Promela, the graph should also look fairly similar as these languages all make use of synchronous communication and since LF is partially based on occam, the transition should not be difficult. In the case of Promela where the sending of one channel over another is allowed, the alias analysis of channels will have to be extended.

To adapt the CCDG to languages such as Java and C will prove more difficult as the process communication does not adhere to a message passing principle and many of the concurrency features are introduced through external library calls that are currently not supported by the CCDG and will therefore require extensive changes.

Model Checking The LF Slicer has been applied to a small example in conjunction with the LF Model Checker. The results were found to be quite favourable and shows promise for the furtherance of such an investigation.

5.3.3.1 Results

Table 11 shows the results when the examples in Listings 4.3 and 4.4 are run through the model checker. The second example is a slice of the first example, taken on line 39 and variable `prod`. As shown by the table, the reduction in all relevant aspects is not insignificant.

Example	States	Loops	MaxDepth	SCCs	Transitions
Listing 4.3	33	23	31	23	55
Listing 4.4	19	12	9	17	30

Table 11: Results when model checking the Product/Sum example

5.4 Concluding Remarks

A program slicer for the LF language was successfully implemented. To accomplish this, a program representation called the CCDG was derived from the PDG to model specific aspects of the language.

The CCDG structure proved useful in reducing the static slicing of a concurrent program to a graph reachability problem. Design decisions were aimed at keeping this as simple as possible and although some of the accuracy may be improved, this solution forms a good base from which to investigate future problems.

Bibliography

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. *ACM SIGPLAN Notices*, 25(6), June 1990.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] Jingde Cheng. Slicing Concurrent Programs – A Graph-Theoretical Approach. In *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 223–240. Springer-Verlag, 1993.

[4] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Elsevier Science, Morgan Kaufmann, 2004.

[5] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh P. Ranganath, Robby Wallentine, and Todd Wallentine. Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 73–89. Springer-Verlag, 2006.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages And Systems*, 9(3):319–349, July 1987.

[7] Leon D. Grobler. A Kernel to Support a Concurrent Language for Verification. Master's thesis, University of Stellenbosch, 2006.

[8] Mark Harman and Sebastian Danicic. Using Program Slicing to Simplify Testing. *Journal of Software Testing, Verification and Reliability*, 5(3):143–162, September 1995.

[9] Mark Harman and Sebastian Danicic. Amorphous Program Slicing. In *WPC*, pages 70–79. IEEE Computer Society, 1997.

[10] Mark Harman and Keith B. Gallagher. Program Slicing. *Information and Software Technology*, 40(11-12):577–582, 1998.

[11] Mark Harman and Robert Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.

[12] Mary Jean Harrold and Gregg Rothermel. Syntax-directed Construction of Program Dependence Graphs. Technical Report OSU-CISRC-5/96-TR32, The Ohio State University, 1996.

[13] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing Software for Model Construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[14] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM TOPLAS*, 12(1), January 1990.

[15] Mizuho Iwaihara, Masaya Nomura, Shigeru Ichinose, and Hiroto Yasuura. Program slicing on VHDL descriptions and its applications. In *Asian Pacific Conference on Hardware Description Languages (APCHDL)*, pages 132–139, 1996.

[16] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[17] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, April 2003.

[18] Jens Krinke. Context-sensitive Slicing of Concurrent Programs. In *ESEC/SIGSOFT FSE*, pages 178–187, 2003.

[19] Jens Krinke. Advanced Slicing of Sequential and Concurrent Programs. In *ICSM*, pages 464–468. IEEE Computer Society, 2004.

[20] Johannes Marais. *Design and Implementation of A Component Architecture for Oberon*. PhD thesis, Swiss Federal Institute of Technology Zürich, 1996.

[21] Lynette I. Millett and Tim Teitelbaum. Slicing Promela and its Applications to Model Checking, Simulation, and Protocol Understanding. In *4th International SPIN Workshop*, 1998.

[22] Lynette I. Millett and Tim Teitelbaum. Channel Dependence Analysis for Slicing Promela. In *PDSE*, pages 52–61, 1999.

[23] Hanspeter Mössenböck and Kai Koskimies. Active Text for Structuring and Understanding Source Code. Technical Report 3, Institut für Praktische Informatik (System Software), Johannes Kepler Universität Linz, Austria, August 1995.

[24] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kauffmann Publishers, Inc., 1997.

[25] Mangala G. Nanda and S. Ramesh. Slicing Concurrent Programs. In *International Symposium on Software Testing and Analysis*, pages 180–190, 2000.

[26] Gleb Naumovich and George S. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel. In *6th International Symposium on the Foundations of Software Engineering*, pages 24–34, November 1998.

[27] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In *7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 338–354, September 1999.

[28] Kurt Nørmak and Kasper Østerbye. Representing Programs as Hypertext. In *Proceedings of Nordic Workshop on Programming Environment Research*, pages 11–24, May 1994.

[29] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *SIGPLAN Notices*, 19(5), May 1984.

[30] Ganesan Ramalingam. Context-Sensitive Synchronization-Sensitive Analysis is Undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):413–430, 2000.

[31] Christoph Steindl. Program Slicing (2): Computation of Data Flow Information. Technical Report 12, Institut für Praktische Informatik (Systemsoftware), 1998.

[32] Christoph Steindl. *Program Slicing for Object-Oriented Programming Languages*. PhD thesis, Johannes Kepler Universität Linz, April 1999.

[33] Riaan Swart. A Language to Support Verification of Embedded Software. Master's thesis, University of Stellenbosch, 2003.

[34] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[35] Frank A. van Riet. LF: A Language for Reliable Embedded Systems. Master's thesis, University of Stellenbosch, 2001.

[36] Shobha Vasudevan and Jacob A. Abraham. Static Program Transformations for Efficient Software Model Checking. In *IFIP Congress Topical Sessions*, pages 257–282, 2004.

[37] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[38] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4), July 1984.

[39] Niklaus Wirth. The Programming Language Oberon. In *Software – Practice and Experience*, volume 18, pages 671–690, 1988.